Neuro 240 Progress Report

Chris GH

Contents

Abstract

This project focuses on using transfer learning to adapt existing machine learning architectures to the visual recognition of cuneiform characters in the Code of Hammurabi (1750 BCE). The focus will be on exploiting the underlying structural regularity of the cuneiform script in order to isolate patterns of strokes, and then on categorizing the stroke layouts into character identifications. Through a combination of ML object-detection and algorithmic segmentation and identification techniques, it is possible to generate stroke-based representations that are distinguished by a second network with high accuracy ($.99\%$); further, given the inherent complexities and ambiguities of any written text, the specific details of hard-to-identify characters and/or sections might offer insight not only into the text, but also into the applicability of AI to character or object recognition tasks more generally. Lastly, the design process for the overall workflow might offer insights into the usefulness of AI in "small data" applications where the dataset on which the network is trained is relatively small; in this case, careful design can be used to introduce context-specific inductive biases that optimize the training process and allow the network to make maximal use of a minimal training set.

1 Introduction & Background

1.1 Introduction

The problem of computer cuneiform character recognition is interesting not only due to the historical significance of the cuneiform writing system, but also due to some of the unique challenges and opportunities the problem poses.

This project will focus on the cuneiform writing used in the Law Code of Hammurabi (CH), one of the most significant cuneiform texts every discovered. Compiled during the reign of the eponymous Babylonian king Hammurabi sometime shortly before 1750 BCE, the law code is engraved into a black stone stele, and consists of a prologue, 300 distinct laws, and an epilogue; these are written in the Old Babylonian language (a dialect of the semitic Akkadian language) using the cuneiform script, and employing a yet-older archaizing writing style. [1]

The cuneiform writing system itself served for over three millenia as the principal writing system of mespoptamia and the surrounding region, and at its peak it served to connect regions as distant as Egypt, Iran, and Anatolia. [2] Developing from gradual simplifications of logographic drawn images via a process not dissimilar to the development of the Latin alphabet from Greek, Phoenician, and ultimately Egyptian precursors (see Fig. 1), the cuneiform system employed arrangements of wedges to represent syllables in the underlying language; however, the same sign could often have multiple possible phonetic values, and numerous signs often had the same phonetic value. In the old babylonian language of the CH, many characters are used both as Akkadian syllabograms and as logographic "Sumerograms," which typically reference the semantic meanings of the same signs in the earlier (and by this period largely obsolete) Sumerian language. [3]

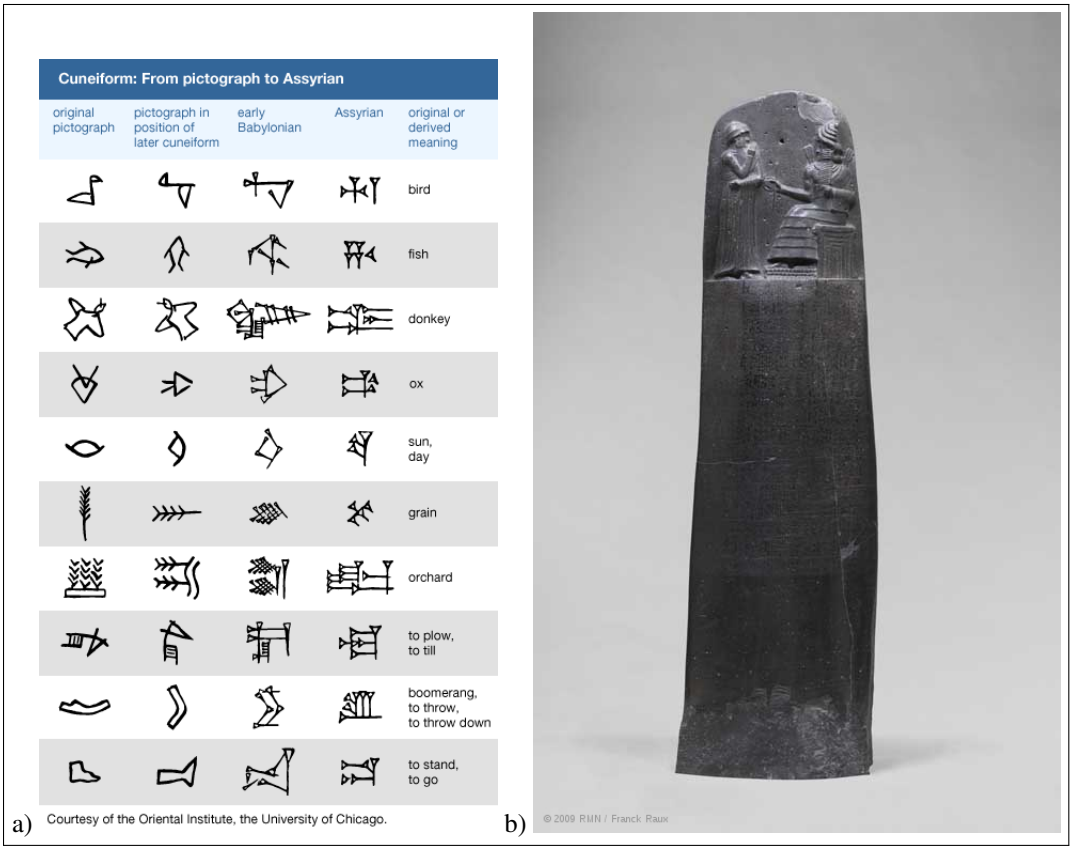
1.2 Literature Review

Past work on computer recognition of cuneiform characters has been fairly limited. Yamauchi et al (2018) reports preliminary steps towards the development of OCR for cuneiform, in the form of a handwritten cuneiform character imageset; however, the finalized imageset has apparently not been published, and the accompanying github repo seems not to have been updated since the paper was published. [4] Nonetheless, the paper provides confirmation of the theoretical possibility of OCR for cuneiform, and in particular despite the challenges inherent to both the writing system and the (fairly non-standardized) way in which hand copy of cuneiform has typically been published; further, the part of the imageset that is available provides a potentially interesting opportunity to test the transfer learning capacity of the final system without requiring additional manual annotation.

Additional work on computer recognition of cuneiform has focused to a large extent on identification of characters from 3-dimensional scans of cuneiform tablets or from high-resolution photographs, both of which are becoming more widely available in the field (however, both would require significant and likely unfeasible amounts of work to re-digitize the numerous texts that have previously been published in hand-copy). Rahma et al (2017) applied OCR techniques to high-resolution photographs of documents using the relatively more standardized Neo-Assyrian script; they also propose using the underlying structure and pattern of wedges, an approach that is adapted for this project. [5] The computer-based 3D tablet imaging tool CuneiformAnalyser uses edge and curvature information in 3D scans of cuneiform tablets to segment individual wedges; [6] while the subsequent steps of sign identification are quite similar to the process attempted here and in Rahma et al (2017), the 3D information used for wedge segmentation are fundamentally different and thus not feasible for the vast majority of cuneiform documents which are not available in 3D form.

More broadly, significant work has been done on the application of OCR to non-latin scripts. One particularly relevant example is the Chinese script, which (like cuneiform) exhibits a very wide range of possible glyphs composed of a smaller set of repeating structural elements that can vary in shape, size, and position; further, like cuneiform, Chinese characters can be quite similar when handwritten, making automated identification more difficult. [7]

In order to improve the efficiency of the model training process, transfer learning from a previously trained model can be applied; this method has been previously investigated in several areas of the broader field of OCR, such as for chinese characters [8] and for early printed books. [9] For this project, previous image recognition algorithms focused on identifying strokes in hand drawings



a) b)

Figure 1: **a)** Development of Cuneiform; from Encyclopedia Brittanica. Of the script styles in the image, the text in the CH is probably most similar to the "old babylonian" characters, but it is nonetheless fairly distinct from these both due to its archaizing lapidary script style and due to its being a monumental stone inscription, rather than a clay tablet. **b)** The Louvre Stele, the most complete surviving copy of the CH; from the Louvre Department of Near Eastern Antiquities. The text of the CH runs vertically in columns below the relief at the top.

might be particularly promising; [10] additionally, reported 30% time efficiency improvements on images from a network pre-trained on text might also present a promising potential methodology that avoids excessive manual labeling. [11]

1.3 Proposed Methodology

The methodology for this project is based loosely on that employed for OCR, alongside the process proposed in Rahma et al (2017). The workflow for an OCR project, generally speaking, consists of preprocessing, segmentation, feature extraction, identification, and postprocessing, each of which are interconnected but largely discrete tasks. [12] These steps, with some modification, are broadly applicable to this project. In particular, I propose to divide the problem of CH cuneiform character recognition into the following six separate tasks, along with the tools/technologies I propose to use to complete them:

1. **Input data augmentation**, i.e. labeling of a training character set from the glossary and separation of column images from plates in Harper (1904) [Manual]

2. **Line segmentation**, i.e. separation of lines of text within columns, with potential intermediate model evaluation based on segmentation correctness [Algorithmic]

3. **Character segmentation**, i.e. detection of individual cuneiform characters within lines of text, with potential intermediate model evaluation again based on segmentation correctness [Object Detection Network 1]
4. **Stroke identification**, i.e. a feature detection step using an object detection network to identify the locations and types of strokes within characters, with potential intermediate model evaluation based on stroke location correctness [Object Detection Network 2]
5. **Character identification**, i.e. visual identification of characters based on the learned stroke layouts, with final evaluation based on identification correctness [Object Identification Network]

This methodology would involve three separate networks, each of which I propose to train via transfer learning from a network trained for a similar task. The final output of the last network will be (hopefully) be the solution to the initial problem, i.e. an ordered list of character identifications over the lines of the text. As pseudocode, the proposed methodology might look like:

```
lines = Line_Segmentation(column_image)
for line in lines:
  chars = Character_Segmentation(line)
  for char in chars:
      stroke_locations = Stroke_Detection(char)
      char_id = Character_ID(stroke_locations)
```

1.4 Limitations

From the outset, one major limitation of the proposed methodology for this particular task is that the distinction between certain minimal pairs of signs is at times somewhat arbitrary; for example, the **ug** and **az** signs are distinguished only by a phonetic complement component that is often left out entirely. In other cases, the distinction is even more unclear: the **ya/ia** sign consists of the **i** sign followed by the **a** sign, but the appearance and phonetic value of **ia** and **i+a** are generally exactly identical. Edge-cases like this indicate that perfect accuracy is not really possible, even under ideal conditions and with a perfectly trained model.

Further, figuring out the actual phonetic value of a given sign requires a degree of natural-language processing not included in the present model. This approach has been pursued successfully for cuneiform word-segmentation and interpretation in other similar contexts (e.g. unicode cuneiform texts [13]); thus, it might in future be possible to incorporate these techniques in order to automate more of the text interpretation process.

Lastly, the manual labeling of input data necessarily leaves significant room for human error. For one, the points chosen to label a given sign element will inevitably be somewhat inexact; even given perfect human labeling accuracy, the variable shapes of the wedges would still introduce some degree of ambiguity. Further, particularly for the script used in the CH, the categories of stroke described in section 2.1 are somewhat imperfect; while the script became more standardized in the later periods, particularly the Neo-Assyrian period, the script used in the CH is much older and exhibits a somewhat broader degree of variation in stroke shape, direction, size, etc. While the categories of Wedge, *Winkelhaken*, and Line are still useful if applied at least somewhat consistently, it is nonetheless useful to keep in mind that they are somewhat arbitrary and inherently unable to account for all observed variation in stroke characteristics.

2 Methods

2.1 Data Acquisition

The first step in the project is the acquisition and labeling of a training and test data-set. Images of the columns of CH text (based on the Louvre Stele) are adapted from Harper (1904) [14]; the same source also contains images of each distinct type of sign appearing in the CH in a glossary. These glossary sign images were separated and labeled manually using Makesense.AI, according to the following labeling conventions:

1. Wedges are labeled as a "line" object (i.e. a pair of points), beginning at the approximate midpoint of the head edge and ending at the point of the tail;
2. Lines (i.e. between wedges) are labeled top-to-bottom along their length;
3. The *Winkelhaken* stroke is labeled from the point of the upper wing to the point of the lower wing, with any reverse-winkelhaken strokes labeled similarly using a separate label type.

The glossary images contain some artifacts from the initial digitization and/or printing of the book itself, as well as some overlap with the line delineators; these are preserved in an attempt to make the model invariant to these artifacts in the wedge-extraction step, but it's important to keep them in mind as a potential source of errors. Additionally, this labeling scheme does not represent information such as the width and shape of the head of a wedge or the location and angle of the middle angle in a *winkelhaken*; while these details can certainly be highly variable between signs, they are not the defining feature in any minimal pair of signs, and therefore do not need to be taken into account at this stage. Examples of the signs as they appear in the glossary and of the labeled sign following manual labeling in Makesense.AI are shown in Figure 2.

The cropped sign images used for the project are contained in the file "Sign_Images.zip", and the manual labels are contained in the file "SL_combined.csv"; these images and labels are used throughout the remaining project steps. The images in the file represent all signs for entries 1 through 58 in the glossary of Harper (1904). This subset of labeled signs is sufficient for the proof-of-concept reported for each step here; however, in order to fully connect the steps into a machine transliteration pipeline capable of parsing an entire section of the CH, it would be necessary to manually label the remaining signs in the glossary.

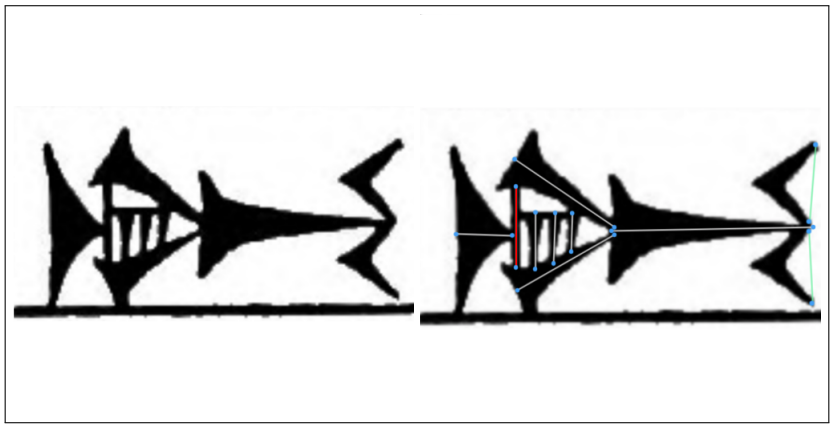


Figure 2: Example Labeled Sign. Above is the *bur* sign (sign 9 in the glossary), along with the same sign manually labeled in Makesense.AI. The sign contains all 3 basic strokes found in the CH signs; in this illustration, wedges are labeled with white lines, *winkelhakens* are labeled with green lines, and a "line" stroke between two wedges is labeled with a red line. Blue dots indicate the label points defining the lines; the coordinates of these points are exported as CSV.

2.2 Line Segmentation

After data acquisition, the next step is segmentation of the column images into individual line images. The segmentation step poses several problems unique to data in the format used here (i.e. printed copy with lines and sign drawn by hand, the format in which many cuneiform documents are most readily available); chiefly, the lines themselves are not exactly straight, nor are they all skewed uniformly. Additionally, the nature of the script is such that a naive algorithm based on the average brightness of a given row of pixels (or even a smoothed average of some number of rows of pixels) will misidentify lines of pixels in the middle of rows of text as line dividers, and will fail to identify the thinnest or most slanted dividers between lines of text. These challenges are illustrated on part of Column 1 of CH in Figure 3.

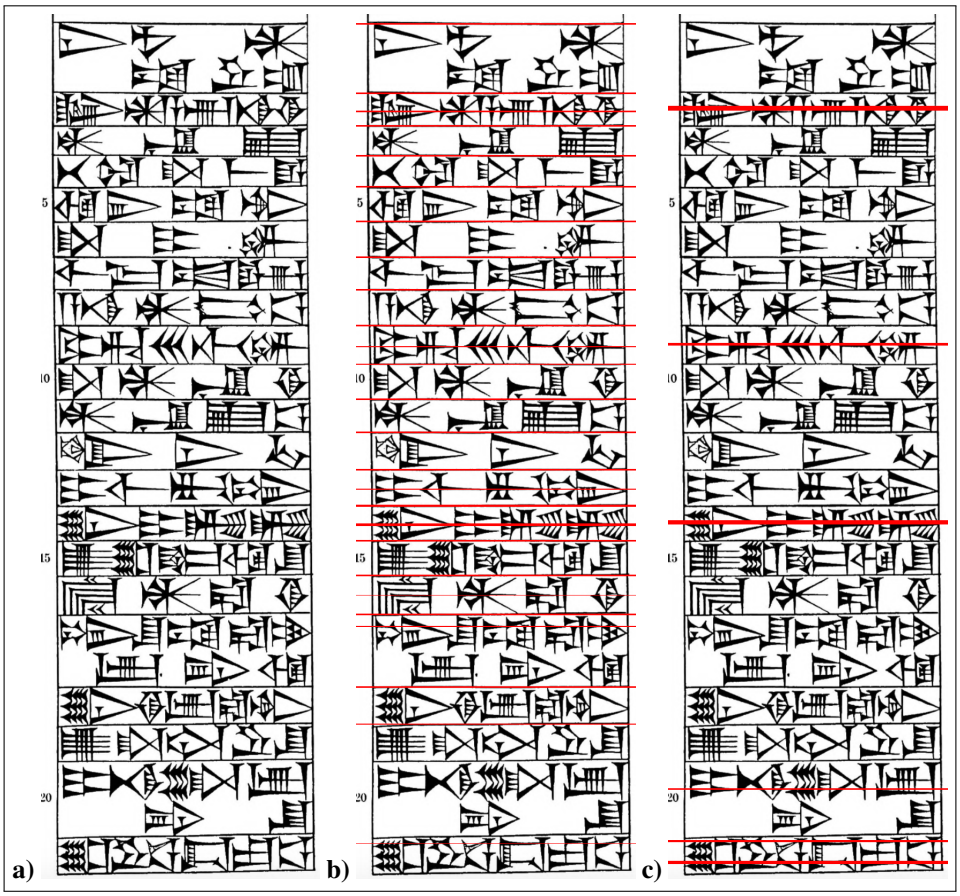


Figure 3: Unsuccessful segmentation algorithms. **a)** Unsegmented image of CH Column 1, part 1, from Harper (1904); initial image dimensions here and in each other case are $1620H \times 522W$, with 3 RGB channels. **b)** Output of segmentation based on the total brightness per row, i.e. the sum over RGB channels of the image values at each pixel, evaluated over each row of the image; red lines are lines of the image where the sum is < 200000. This algorithm does not highlight every divider line (for example, the dividers below text lines 19, 20, and 21 are unhighlighted), and it incorrectly highlights lines of pixels in the middle of text lines (for example, lines 2, 9, etc.). This indicates that this method cannot be made to work by raising or lowering the threshold (i.e. it is not possible to raise the threshold to exclude the text lines without also erroneously excluding more divider lines). **c)** Output of segmentation based on the average brightness over 5 consecutive rows; red lines are lines where the average sum of rgb values across the rows is < 230000. Here, no divider lines are correctly isolated, but several lines in the middle of text are incorrectly isolated; as before, this indicates that no threshold value will correctly separate dividers from text lines.

Ultimately, these issues were resolved using a 3-step approach: Convolutional Filtering, Gap-Checking, and Windowing. In the convolutional filtering step, a 1-d horizontal edge detection filter is run along the length of the image; the output is positive at the top of a horizontal edge, and negative at the bottom of the edge (small note: currently, my implementation is technically 2d, but I'm summing horizontally anyways; I'll plan to actually switch it to 1d at some point while I'm refining the segmentation process overall, since it's a bit slow/inefficient to iterate over each pixel, esp. if it's not necessary). The Gap-detection step acts essentially as a filter on top of the first filtering step; at each point in the image, all pixels in the union of a pair of smalls windows are checked, and if both windows are totally white (i.e. all pixel values above a set threshold, 200), then the line is removed. This step is highly effective, but at times it's a bit *too* effective, since the dividing lines themselves sometimes have small gaps in the original due to printer error or because they're hand-drawn; these gaps are the reason for using two overlapping windows: one shorter but wider, and one tall and

narrow. This combination of windows does a decent job in differentiating between artifact gaps in dividers and actual gaps; however, the hyperparameters of window size require a fair bit of tuning. Lastly, a minimum distance between any two dividers is imposed. In the current implementation, this involves first generating an upper and lower bound from each "cluster" of dividers; this is done by starting once from the top of the image, and once from the bottom, exploiting the clustered output of the previous steps (however, it could in theory break if the previous steps are unable to eliminate lines in the middle of text). The upper and lower bounds for each divider can be averaged to obtain a middle-point guess to divide the image into totally discrete segments; alternatively, by using upper and lower bounds for the top and bottom of each segment, the image can be divided into overlapping segments that help to ensure parts of signs are not cut off if they go close to or slightly over the dividing line. The results of this process for part of Column 1 of CH are shown in Figure 4.

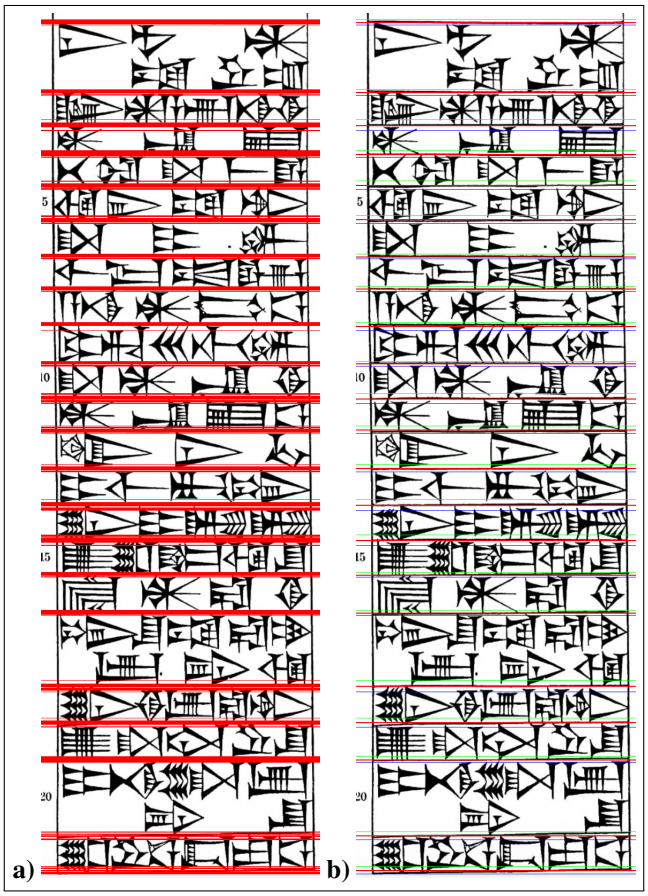


Figure 4: More successful segmentation algorithms. **a)** This segmentation output implements the first two steps of the workflow discussed, namely (1d) convolutional filtering and gap-detection. The gap-detection windows used were $5W \times 10H$ and $1W \times 20H$. **b)** This segmentation output applies the windowing step to the previous segmentation output. The min window used here is 40; the red lines represent the midpoint guess for each divider, and the green and blue lines represent the upper and lower bounds, respectively. Midpoint line thickness is increased to 2px; the upper pixel in each line is the actual midpoint.

Following the midterm report, the Line Segmentation step was condensed into a single, self-contained function, contained in the file "240_Segmentation.ipynb"; the function takes as input a single .png image file of one column segment from CH (i.e. half of one plate from Harper 1904), and returns the lines extracted algorithmically from that column.

Although this step does not actually contain any direct machine learning, the process of visual convolution (using a filter in one dimension) used to identify the divisions between lines is

certainly similar to and inspired by the feature-detection strategies used by ML image processing networks. Further, the line segmentation step can be thought of as essentially an inductive bias for later steps; algorithmically dividing the image into lines serves to bake in the assumption that no character or stroke will cross between the boundaries between lines, simplifying the character/stroke-segmentation and character ID steps.

2.3 Character Segmentation

Character segmentation was performed using the TorchVision module from Pytorch; the object detection model was trained from a pre-trained Faster_RCNN network with a Resnet-50 backbone (fasterrcnn_resnet50_fpn), with a custom database class following TorchVision specifications for model data. The model was trained using line images segmented from column images as in the previous section, with boxes annotated around each sign in MakeVision.ai and saved to CSV format; this format was then converted to numpy array and stored directly in a TorchVision-friendly database. The model was trained for 10 epochs, with 5 line images reserved as the test set and the remainder as the training set (lines were shuffled prior to train-test splitting). All code for this section is provided in the file "240_SignDetector.ipynb."

Results for character segmentation are provided below, in figure 2. While the segmentations are not perfect, they are generally successful, especially when signs are well-separated. Further, the network is able to pick up on some important but nontrivial inferred facts about sign placement: firstly, that signs can overlap horizontally and/or vertically (for example, the first two signs in the first line, or the wecond and fourth signs in the third line); secondly, that signs might not be clearly separated (e.g. the first two signs in test line 2); and thirdly, that signs might have internal gaps (e.g. sign two in test line 1). Although the model still makes some mistakes where divisions are unclear, such as at the end of test line 1 or the beginning of test line 4, the relative success on such a small training set indicates that this method of character segmentation shows significant progress for this application. Although data acquisition/augmentation is still something of a bottleneck for more general applications of this methodology, these results indicate that it likely will not pose major issues in terms of the broader usefulness of the project.

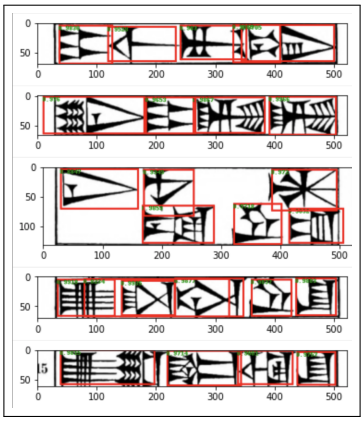


Figure 5: Character Segmentations of the 5-line test set, with threshold score ≥ 0.8.

2.4 Stroke Segmentation

Stroke segmentation was performed using a modified Keypoint_RCNN object detection network capable of learning bounding boxes and keypoint locations, as well as another custom database class following TorchVision specifications for model data. Keypoint_RCNN is a modification of the faster_RCNN network available in the Pytorch model zoo; in this case, the model was used with a pretrained Resnet-50 backbone. However, because the stroke representation used here contains 3 classes of strokes, it was necessary to modify the classes in the Keypoint_RCNN output; as a result, it was not possible to fully exploit transfer learning from a pretrained Keypoint_RCNN network.

The stroke segmentation network was trained for 50 epochs; the initial average bbox precision was 0.00, and the final average bbox precision was 0.245 (for IoU=0.50:0.95 and MaxDets = 100, the default first reported loss value for the optimizer used by the Keypoint_RCNN network). The model was trained on the entire set labeled in the Data Augmentation step, minus 10 images reserved as a test-set. Despite the relatively small size of the training set (98 labeled images in total), the model was able to learn meaningful representations of the strokes that make up cuneiform signs, as shown below.

Sample outputs from the stroke segmentation network are shown and discussed in Figure 6.[1] The code for this section of the project is contained in two files: "240_StrokeSeg_Trainer.ipynb" and "240_StrokeSeg_Runner.ipynb". The "Trainer" ipynb file contains the code used for the initial setup and training of the model, as well as some visualization tools; at the end of the file, the model's state dict is exported via ONNX; this allows the model to be loaded without re-training, a necessary step due to the time and computational resources required each time the model is trained (as well as the limited GPU resources available in google Colab). The "Runner" ipynb file contains code required to load the model and run it to generate predicted keypoints for the labeled data set; these predictions serve as the input for the Character Identification step. The final output for the step is a list of one python dict per image in the labeled set, containing predictions of keypoints and labels as well as the scores assigned by the model. These predictions were saved as a pickle file and used as part of the input for the Character Identification network.

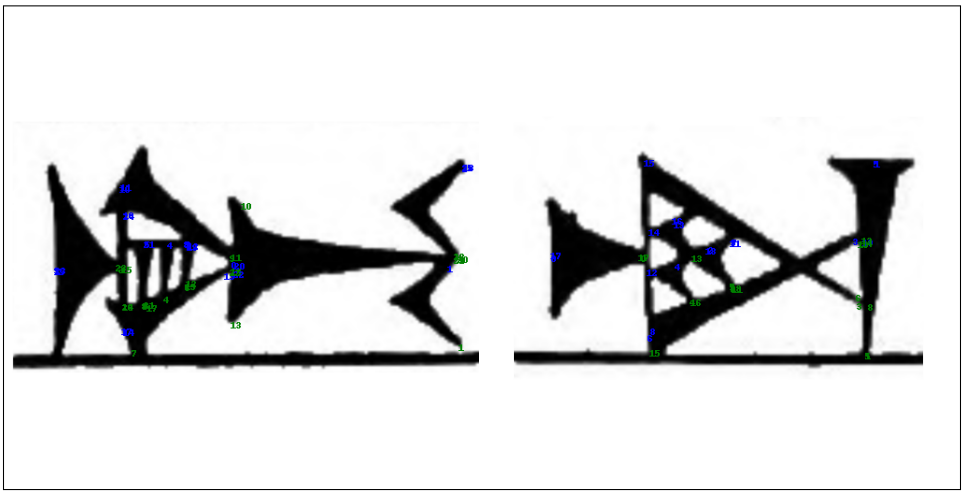


Figure 6: Stroke Labels corresponding to a subset of the learned keypoints output by the Stroke segmentation network, for the cuneiform signs **bur** and **ÌR**. Here, each pair of associated points has the same number; blue numbers are the start/head of a stroke, and green numbers are the end/tail (following the same labeling convention as described in the Data Acquisition section)

2.5 Character Identification

Character identification was implemented as essentially a matching task: a network was trained to compare predicted keypoints with ground-truth labeled keypoints, and determine whether the two keypoint representations corresponded to the same sign. This task was performed using a novel (but fairly simple) feedforward pytorch neural network with the following architecture:

```
self.flatten = nn.Flatten()
self.linear_relu_stack = nn.Sequential(
        nn.Linear(100*6 + 50*6, 512),
```

[1]There's unfortunately not really any clear way of showing what exactly is going on here, since the strokes overlap and the labels occlude each other in any image that isn't unreasonably blown-up; hopefully the numbers are still clear enough to get the general point of the network across. Not shown in the diagram are the predicted labels; these were also included in the final prediction output.

```
nn.ReLU(),
nn.Linear(512, 512),
nn.ReLU(),
nn.Linear(512, 128),
nn.ReLU(),
nn.Linear(128, 128),
nn.ReLU(),
nn.Linear(128, 2))
```

The 6 input dimensions correspond to the X and Y coordinates of the two keypoints for each stroke in the predictions output from the previous step, the bbox score for the stroke (measuring the confidence of the previous model), and the stroke type label (1 for wedge, 2 for winkelhaken, and 3 for line). The input points for the training step consisted of a 900-entry array containing up to 100 6-element "slots" for a learned keypoint and up to 50 for a ground-truth keypoint; each stroke is represented by 6 elements, so this allows the model to take up to 100 stroke guesses from the previous step and up to 50 ground-truth strokes. The Character identification network was trained for 20 epochs; the model training curves are shown below in figure 7.

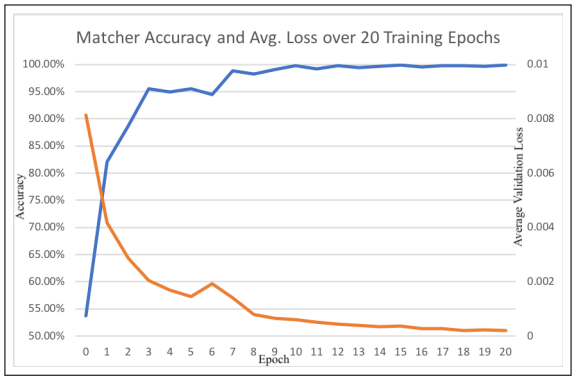


Figure 7: Training curves for the Character Identification model, showing the decrease in average loss and increase in accuracy over the course of the 20 training epochs. The final accuracy was 99.9%, indicating a high degree of model convergence.

Since the Character Identification step is the final step of the overall pipeline as currently implemented, sample results and discussion are presented in more detail in the following section. Of note in terms of the technical details of the step itself, however, is the fact that the input representation here is much simpler than the initial image data: the images are padded/scaled to 500×500 pixels, for 250,000 potential input dimensions, but the actual inputs to the final model are far more lightweight, at only 900 dimensions (many of which are padded and actually empty).

Code citations for each section are included in the code files; these consist largely of references to Pytorch documentation and tutorials. All code was developed and run in .ipynb notebook files in google Colab, with the exception of the (local) parser used to extract loss values from the text output of the Character Identification network's training steps.

3 Results and Future Directions

Some interesting results obtained from the final step are reported below, in Figure 8. The format for the data is as follows: the Model is run in evaluation mode on the learned keypoint representation for Sign 1 from the stroke-segmentation step and the ground-truth labeled keypoints for Sign 2, and the (softmax) model output represents the model's best guess as to whether the representations are of the same sign (1) or different signs (0). Due to the softmax function applied to the raw logits produced by the model, the outputs range from 0 to 1, and always sum to 1.

Sample Model Results, on ID and OOD data			
Sign 1	Sign 2	(Softmax) Model Output	
		0 (different)	1 (same)
1.0	1.0	0.02705	0.97295
11.0	11.0	0.19490	0.80510
26.0	26.0	0.30852	0.69148
1.0	2.0	0.96402	0.03598
2.0	1.0	0.99895	0.00105
9.0	10.0	1.00000	0.00000
13.0	14.0	0.90605	0.09395
41.0	42.0	0.97729	0.02271
28.3	25.0	0.00042	0.99958
25.0	28.3	0.99696	0.00304
54.0	54.1	0.08210	0.91790
49.0	49.1	0.52932	0.47068

Figure 8: Some interesting results from the Character Identification step; discussed in more detail above/below.

Of note: in general, the network is able to identify representations for the same sign, as is the case for signs 1.0, 11.0, and 26.0 (the lowest model confidence for comparison of the same sign); however, this is not always the case for comparisons of different signs, and in some cases (such as the ground-truth keypoints for sign 28.3 and the learned representation for sign 25.0) the model has a high confidence in an incorrect conclusion. Also of note, though not unexpected, is the non-commutativity of the model: the results for model comparison of one sign's learned representation to a second sign's ground-truth keypoints are not in general the same as the predictions for the ground-truth keypoints of the first sign and the learned representation of the second, as occurs above with signs 1.0 and 2.0, and more drastically (and incorrectly) for signs 25.0 and 28.3 (the most incorrect model guess reported on the permutations of the labeled data set). However, for fairly dissimilar signs (such as 9 and 10) the model is easily able to distinguish the two representations; even for fairly similar signs, sich as 13.0 and 14.0 (which differ only by one small stroke in the middle of the sign), the model is generally able to distinguish the two with reasonably high certianty. With more training and data, it is likely that the remaining incorrect cases could be largely eliminated; however (as discussed above in the Limitations section) it is likely not possible to eliminate every incorrect identification, due to ambiguity inherent in the writing system. Lastly, the table also includes some reported results on out-of-distribution data: the non-matching input samples (i.e. those with label 0) were chosen such that the two signs were not only different in form, but also in underlying identity; that is to say, pairs of different forms of the same sign were not used in the training process, so the model has not seen these pairs before. The model is still able to learn the differences between these forms in some cases (e.g. 54.0 and 54.1, which differ in the thickness and length of a stroke in the middle of the sign); however, in other cases the output is fairly close to random chance (e.g. 49.0 and 49.1, which differ in the presence or absence of a small vertical stroke in the lower left corner of the sign).

In conclusion, the most significant results of the project are (in my opinion) that it is possible to achieve meaningful progress towards machine transliteration of cuneiform signs using only a very limited training set, and that it is possible to use baked-in inductive biases of underlying stroke structure to drastically simplify the input space for the final character identification network. Further work would most immediately focus on labeling and training on the remainder of the glyphs occuring in CH, in order to obtain a full pipeline for machine transliteration. Subsequent work might focus on more technical details, such as the role of model architecture details in model performance in the character identification step. Finally, it might be interesting to look into transfer learning from the models developed here for machine transliteration of other cuneiform documents, or even other scripts. Ultimately, while this project did not strictly produce a full machine transliteration of the

CH, it was still successful in demonstrating the feasibility of each step needed to do so despite a small labeled training dataset and limited computational resources, and it succeeded in developing and testing each step of the pipeline needed for project focused less on the technical details of the underlying ML process and more on simply producing a full transliteration of the CH. Given the relative scarcity of other work applying machine learning to cuneiform text transliteration (and the apparent total lack of any prior work applying machine learning to the script style used in the CH), there's still significant room for promising future work to be done in order to further refine and expand on the initial process and steps presented here.

References

[1] Roth, Martha T. *Law Collections from Mesopotamia and Asia Minor*. Edited by Piotr Michalowski. 2nd ed. WAW 6. Atlanta: Scholars Press, 1997.

[2] Radner, Karen, and Eleanor Robson, eds. The Oxford handbook of cuneiform culture. Oxford University Press, 2011.

[3] Huehnergard, John. A grammar of Akkadian. Brill, 2018.

[4] Yamauchi, Kenji, Hajime Yamamoto, and Wakaha Mori. "Building a handwritten cuneiform character imageset." *Proceedings of the Eleventh International Conference on Language Resources and Evaluation* (LREC 2018). 2018.

[5] Rahma, Abdul Monem S., Ali Adel Saeid, and Muhsen J. Abdul Hussien. "Recognize Assyrian cuneiform characters by virtual dataset." *2017 6th International Conference on Information and Communication Technology and Accessibility (ICTA)*. IEEE, 2017.

[6] Fisseler, D., et al. "Towards an interactive and automated script feature analysis of 3D scanned cuneiform tablets." Scientific Computing and Cultural Heritage (2013): 16.

[7] Yin, Yue, et al. "Deep learning-aided OCR techniques for Chinese uppercase characters in the application of Internet of Things." *IEEE Access 7* (2019): 47043-47049.

[8] Tang, Yejun, et al. "CNN based transfer learning for historical Chinese character recognition." *2016 12th IAPR Workshop on Document Analysis Systems (DAS)*. IEEE, 2016.

[9] Reul, Christian, et al. "Transfer learning for OCRopus model training on early printed books." *arXiv preprint* arXiv:1712.05586 (2017).

[10] Wu, Xingyuan, et al. "Sketchsegnet: A rnn model for labeling sketch strokes." 2018 IEEE 28th International Workshop on Machine Learning for Signal Processing (MLSP). IEEE, 2018.

[11] He, Yang, Jingling Yuan, and Lin Li. "Enhancing RNN based OCR by transductive transfer learning from text to images." Proceedings of the AAAI Conference on Artificial Intelligence. Vol. 32. No. 1. 2018.

[12] Boulid, Youssef, et al. "Handwritten Character Recognition Based on the Specificity and the Singularity of the Arabic Language." *International Journal of Interactive Multimedia and Artificial Intelligence*, vol. 4, no. 4, 2017, p. 45.

[13] Gordin, Shai, et al. "Reading Akkadian Cuneiform Using Natural Language Processing." *PloS One*, vol. 15, no. 10, 2020, p. e0240511.

[14] Harper, Robert Francis. *The Code of Hammurabi, King of Babylon, about 2250 B.C. : Autographed Text, Transliteration, Translation, Glossary, Index of Subjects, Lists of Proper Names, Signs, Numerals, Corrections, and Erasures, with Map, Frontispiece and Photograph of Text.* 2d ed., University of Chicago Press, 1904.