CHINESE CHARACTER RECOGNIZER

Alec Santiago 47853661 `alecas@uci.edu`

Taylor Fialkowski 22994791 `tfialkow@uci.edu`

Aleksa Kostic 52228237 `akostic@uci.edu`

ABSTRACT

Writing Chinese characters can be a rigorous task, especially for those who are non-native to the language; a slight change of stroke in writing could lead to a different character. Simply doing an eyeball comparison of handwritten Chinese characters to a standardized Chinese character could help in character writing practice. However, having an automated process would hasten the learning process. To close this gap is the purpose of our project, where we ultimately aim to use computer vision and convolutional neural networks to show how close handwritten characters are to actual characters. Through the use of a GUI, Python's Numpy, OpenCV, PyTorch, and a collection of over 5,000,000 images of handwritten characters, we develop a system that can assist a learner of the Chinese written language in practicing, comparing, and improving their writing accuracy.

1 INTRODUCTION

Computer vision has been utilized by numerous technologies, from self-driving cars to medical-image recognition. What we are trying to accomplish is utilizing computer vision as a learning method by applying both image detection and a graphical user interface (GUI) with which a user can attempt to draw Chinese characters.

We utilize the GUI to allow users to freely write any Chinese character. Once they are done writing, there is a frame to show what our program thinks the character is and the likelihood of the handwriting being the predicted character. This program utilizes a convolutional neural network model where our input is the GUI-handwritten Chinese characters, and the output is the expected Chinese character. The model is trained with various images of handwritten Chinese characters, and utilizes a training and testing split to assess its accuracy. An overview of how the whole system work is as follows: take the image from the GUI, pass it to the CNN model, output the expected Chinese character and its likelihood. These are the fundamental building blocks toward innovating mobile language products.

2 PROJECT RELATED WORKS

The handwritten Chinese character recognizer is closely related to the handwritten digit recognition using the MNIST data set.[1] Instead of the MNIST dataset, we will use a dataset from Kaggle called "Handwritten Chinese Character (Hanzi) Dataset", which draws from the CASIA Online and Offline Chinese Handwriting Databases.[2] The zipped dataset for the Chinese characters is about $14\,\mathrm{GB}$ which includes all characters, not just the English alphabet.[3] Due to the large size of the dataset, the model must be optimized to handle large numbers of classes for classification. Additionally, since our CNN like many others assumes that all input image size are equal,[4] we will preprocess each image by first trying zero-padding rectangular images up to square images in the appropriate dimension. This should improve the accuracy by avoiding loss of features, prevent label mismatching, and this will also improve the efficiency of our model.[5]

Another similar project is from the HandWritten Digit Recognition GUI App, where they utilized OpenCV and Tkinter to create a GUI method that takes in a user's digital drawing as input, and upon submission predicts and outputs the written digit with a likelihood between 0 and 1.[6]

The main issue is being able to detect language in certain backgrounds. One way to handle this is by having a standardized background to detect the language; however, what our goal is to be able to live-scan a background to detect the chinese characters. There are two main libraries involved with this, Tesseract OCR and Google Vision. First to note, Tesseract OCR is free to use, while Google Vision is a paid subscription. An article[7] talks about the performance of detecting handwritten text of children with Tesseract OCR and Google Vision. The article shows how the performance of Google Vision is far superior compared to Tesseract, and due to the complexity of Chinese characters, we might consider utilizing Google vision to conduct benchmark comparisons.

3 Data Description

The Kaggle Handwritten Chinese Character (Hanzi) Dataset contained *(6,800 classes at 1600 images)* 11,008,000 training images and *(6,880 classes at 400 images)* 2,752,000 testing images of handwritten Chinese characters of various size. The zipped data is 13 GB in size. After unzipping and cleaning the data, it was about 9 GB in size, and after resizing the images, the data was roughly 12 GB in size.

The first issue was a matter of the labeling of the folders (18 GB). As the folder labels were in Mandarin Chinese, downloading the dataset meant encoding errors as the labels were converted to Latin encoding upon download, but were originally encoded with GB2312 encoding. So, using Python, we encoded the new, Latin-character labels to GB2312 and then decoded back to UTF-8 to obtain Mandarin character labels.

The second issue to address in preprocessing was eliminating photos that did not correspond to the correct label. Upon inspecting a subsample of 50 folders, we found that the incorrectly labeled photos always followed the same label pattern; the naming format for the correct photos was of the type `*.png`, where * was any numeric character, and that of the incorrect photos was of the format `*_*.png`.

```
Get-Childitem -path ~\data\test -Filter *.png -Recurse |
 where-object {$_.Name -ilike "*_*"} | Remove-Item -Force

Get-Childitem -path ~\data\train -Filter *.png -Recurse |
 where-object {$_.Name -ilike "*_*"} | Remove-Item -Force
```

Figure 1: PowerShell code for cleaning up images in incorrect directories.

Using PowerShell and the commands in Figure 1, we were successful is removing all of the incorrect data. We remained with *(6,800 classes at 800 images)* 5,504,000 training images and *(6,880 classes at 200 images)* 1,376,000 testing image (9 GB).

The following issue had to do with image size. Although the images were all black and white, which allowed for a starting input channel of size one (as opposed to three for RGB images), the CNN that we were building required that inputs be of the same size. While interpolation of images was a common practice, it was noted that changing the size of an image by stretching could deform essential patterns in the natural image. So, we chose to add "zero-padding" to our images as a study showed that "zero-padding had no effect on the classification accuracy but considerably reduced the training time".[5] This augmentation of data was supposedly a safe method shown to assist in the building of CNN models, and for our dataset, didn't increase the size of the data too drastically (9 GB to 12 GB).

A primary issue in our data size, however, was the our models were not able to train fast enough, with training estimates stretching past 72 hours, and that even after hundreds of epochs and multiple days of training, testing accuracies remained abysmal, usually staying below 1%. As such, we learned

```
Get-Childitem -path ~\data\train -Directory -Recurse |
where-object -FilterScript { $_.Name -notin
("的","一","是","了","我","不","人","在","他","有","这","个","上",
"们","来","到","时","大","地","为","子","中","你","说","生","国",
"年","着","就","那","会","家","可","下","而","过","天","去","能",
"对","小","多","然","于","心","学","么","之","都","好","看","起",
"发","当","没","成","只","如","事","把","还","用","第","样","道",
"想","作","种","开","美","总","从","无","情","己","面","最","女",
"但","现","前","些","所","同","日","手","又","行","意","动","方",
"期","它","头","经","长","儿","回","位","分","爱","老","因","很",
"给","名","法","间","斯","知","世","什","两","次","使","身","者",
"被","高","已","亲","其","进","此","话","常","与","活","正","感",
"见","明","问","力","理","尔","点","文","几","定","本","公","特",
"做","外","孩","相","西","果","走","将","月","十","实","向","声",
"车","全","信","重","三","机","工","物","气","每","并","别","真",
"打","太","新","比","才","便","夫","再","书","部","水","像","眼",
"等","体","却","加","电","主","界","门","利","海","受","听","表",
"德","少","克","代","员","许","先","口","由","死","安","写","性",
"马","光") } | Remove-Item -Recurse -Force


Get-Childitem -path ~\data\test -Directory -Recurse |
where-object -FilterScript { $_.Name -notin
("的","一","是","了","我","不","人","在","他","有","这","个","上",
"们","来","到","时","大","地","为","子","中","你","说","生","国",
"年","着","就","那","会","家","可","下","而","过","天","去","能",
"对","小","多","然","于","心","学","么","之","都","好","看","起",
"发","当","没","成","只","如","事","把","还","用","第","样","道",
"想","作","种","开","美","总","从","无","情","己","面","最","女",
"但","现","前","些","所","同","日","手","又","行","意","动","方",
"期","它","头","经","长","儿","回","位","分","爱","老","因","很",
"给","名","法","间","斯","知","世","什","两","次","使","身","者",
"被","高","已","亲","其","进","此","话","常","与","活","正","感",
"见","明","问","力","理","尔","点","文","几","定","本","公","特",
"做","外","孩","相","西","果","走","将","月","十","实","向","声",
"车","全","信","重","三","机","工","物","气","每","并","别","真",
"打","太","新","比","才","便","夫","再","书","部","水","像","眼",
"等","体","却","加","电","主","界","门","利","海","受","听","表",
"德","少","克","代","员","许","先","口","由","死","安","写","性",
"马","光") } | Remove-Item -Recurse -Force
```

Figure 2: PowerShell code for fast data deletion.

that we needed to use a subset of classes if we at least wanted to get some preliminary testing and results. So, we sub-sampled 190 classes for the CNN and 25 classes for the FFNN using the code in Figure 2.

This data contained *(190 classes at 140 images)* 26,600 testing images and *(190 classes at 600 images)* 114,000 training images, approximately 600 MB in combination.

However, after creating our machine-learning models and integrating the models with the GUI, it became apparent that zero-padding did in fact pose an issue for our prediction purposes. This paper asserts that while zero-padding may not have played a significant role for images with a multitude of different features, such as colors and real-world objects other than lines on paper, for our images, adding zero-padding registered in the models as "useful" features, specifically, more black pixels to

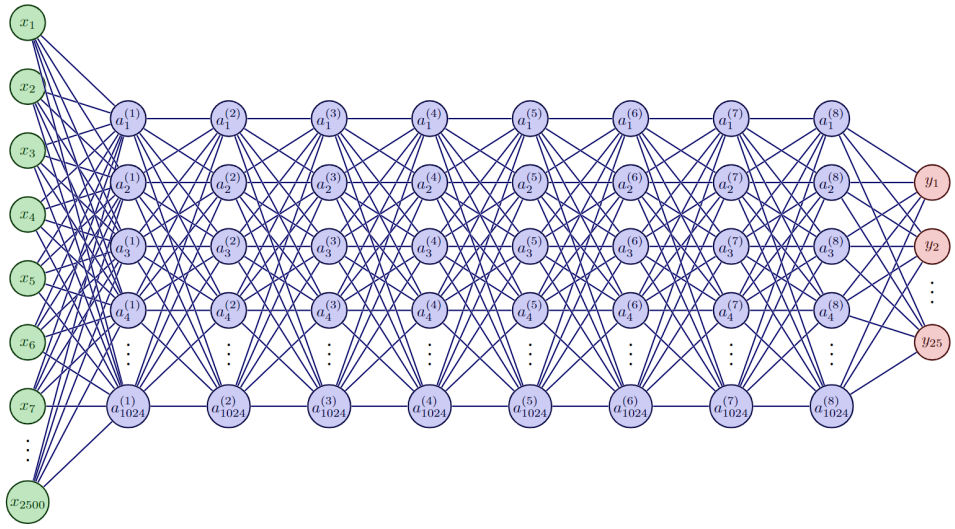

Figure 3: Feed-forward neural network architecture diagram.

count toward the predictions. The problem with this in combination with our GUI was that when we captured drawings for prediction, the capture was of the entire canvas, which had a white background and no zero- or black-padding. As such, the initial rounds of testing drawing predictions yielded in nearly 0% prediction accuracy despite having models with supposed 82% prediction accuracy.

To fix this issue, we wrangled the data again from the original zip file, this time simply resizing the images via interpolation rather than adding any padding, increasing the size to approximately 666 MB, and after training the models and re-implementing them to the GUI, we yielded much more accurate predictions from drawings.

4 Approach

4.1 Feed-Forward Neural Network (FFNN)

We began neural-network modeling using a simple, linear, feed-forward network architecture. The parameters were gathered and modified by measuring the performance of models with various combinations of parameters (i.e. from different optimizers, hidden layers, etc). Starting from a model using the Adam optimizer, and with four hidden linear-combination layers.

We explored different numbers of hidden layers, primarily sticking to linear combinations, in addition to different numbers of prediction classes. Sticking with the Adam optimizer for the whole of the project, we found that increasing the number of layers would improve accuracy up until about a depth of eight layers as shown in Figure 3. Additionally, adjusting the size of the layers varied accuracy, where larger-sized layers did well until exceeding 2,048 features. We found a significant improvement upon introducing a dropout layer before the final linear activation layer leading to the prediction output. Testing the addition of another dropout layer in the middle of the forward feed, we saw very little improvement and opted to remove the additional dropout layer. Other layers types (activation functions) have yet to be considered and reviewed.

Another combination we attempted was changing the learning rate from 0.001 (1×10^{-3}) to 0.0005 (5×10^{-4}).

4.2 Convolutional Neural Network (CNN)

For our primary model, we opted to create a CNN model. Like in the FFNN model, the parameters were gathered and modified by measuring the performance of models with various combinations of parameters (i.e. from different optimizers, hidden layers, etc). Starting from a model using the Adam optimizer, and with four hidden sequential layers, eventually using five hidden layers as shown in

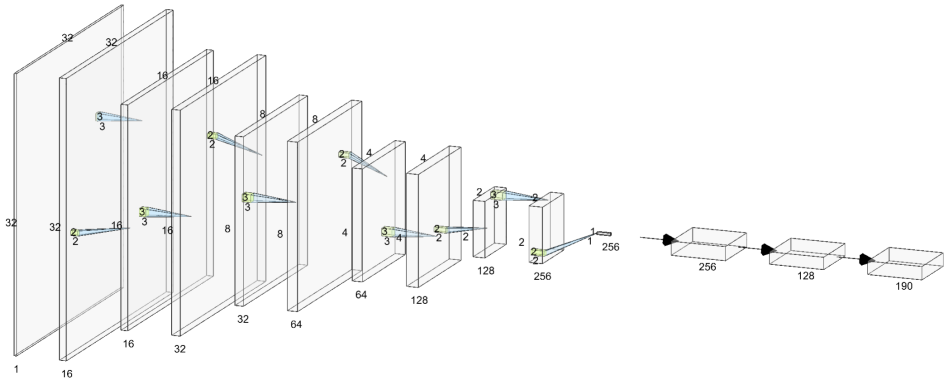

Figure 4: Convolutional neural network architecture diagram.

Figure 4. Using the following formula: output channels $= \left(\frac{\text{input height} + \text{kernel width} + 2*\text{padding width}}{\text{stride}}\right) + 1$, we used "same-padding" for the convolutional layers, in which the kernel size was 3×3, and padding and stride were of size 1, and dilation was standard. Each convolutional pass, minus the first one, doubled the number of channels in the output (i.e. 1 to 16 to 32, 32 to 64, and the finally to 128). After convolution, we applied batch normalization, followed by ReLU activation. For the number of outputs for the max pooling, we went with a formula nearly identical to the one above, but that kernel size was 2×2, padding was also 2×2, and everything else was held by standard.

As the project stands, we explored different numbers of hidden layers, as well as types of hidden layers, in addition to different numbers of prediction classes. Sticking with the Adam optimizer for the whole of the project, we found that there was a slight improvement in model performance using five hidden layers – where the layer structure remained constant of the type: convolution, normalization, ReLU activation, and max pooling. Additionally, we found an slight improvement upon introducing a dropout layer before the final linear activation layer leading to the prediction output. Other layer designs as well as numbers have yet to be considered and reviewed. Another combination we attempted was changing the learning rate from 0.001 (1×10^{-3}) to 0.0005 (5×10^{-4}), though we did not see significant improvements in performance.

Accuracies in this model started off relatively high earlier on in training, ranging from about 65%-70% after the second epoch. Within twelve epochs, we saw the model begin the plateau in both its training accuracy at around 85% and in its loss at around 0.8. *See evaluation section for more details.*

4.3 Graphical User Interface (GUI)

The GUI, implemented with Tkinter, had a set-sized black and white canvas. Given a drawing on this canvas, the unique combination of pixels could potentially be recognized as a Chinese character. We created the main window of the application, along with a canvas on which a user could draw, and buttons for the user to send the drawing for prediction, or for the canvas to be cleared. Once the user finished drawing, they would able to select a button named "classify" and, once selected, would start the character prediction. Upon pressing the button, the canvas was captured via ImageGrab from the Python Image Library, or PIL. Once the image was grabbed, the image would be saved, and then reloaded. The reloaded image was then passed through the PyTorch prediction model, first passing through PyTorch's transform functions such as resizing and re-channeling, and then passed through the prediction model, after which the predicted class along with its likelihood were passed back to the GUI and reflected in the user window. Some examples of the GUI after output are shown in Figure 5.

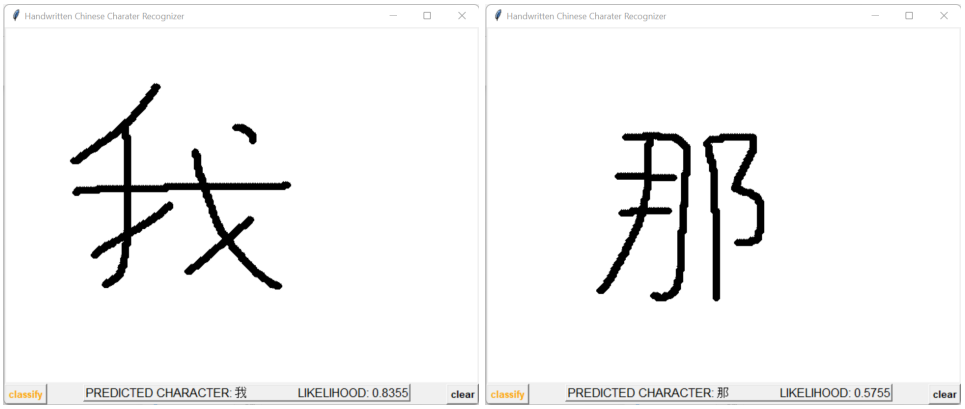

Figure 5: Two screen captures of our GUI after pressing "classify". Both drawn characters were predicted correctly.

5 Evaluation Results

5.1 Model Evaluations

Starting with a simple FFNN with an original learning rate of 0.001, we found that testing accuracy was unstable, and changing the number of layers or the size of the layers didn't make a large difference in the stability of our accuracy. Another combination we attempted was changing the learning rate from 0.001 (1×10^{-3}) to 0.0005 (5×10^{-4}). This prevented the model from sporadically and frequently having extremely high loss and extremely low accuracy in the middle of model training – such as in the twenty-eighth epoch out of fifty. For example, when the learning rate was at 1×10^{-3}, we saw model accuracy plummet from roughly 60% down to 0.05% and then very slowly crawl back up; when we used a learning rate of 5×10^{-4}, however, this "random" jump did not occur nearly as frequently.

The FFNN however failed to yield accuracies higher than 65% despite different combinations of layers, epochs, and learning rates that we attempted up to this point.

When we originally tried to implement the feed-forward neural network, we tried to use all 6,880 classes with the entire valid dataset of images. However, we found that training just the FFNN alone would take over 72 hours. So, we decided to use a subset of classes, specifically 190 classes. However, even this amount was too much for the FFNN, but this time in terms of accuracy. The accuracy for the FFNN when using 190 classes remained at less than 1% even after 50 or 100 epochs of training. So, we used just 10 classes to train the FFNN. As this proved to provide okay accuracy of around 50% to 60%, we bumped up the number of classes to 25. After playing around with the layering of the FFNN in terms of the number of layers, the size of the layers, and the inclusion of a Dropout layer, this FFNN model defined in this code was deemed the most optimal for the sake of starting off our project. The loss, however, would skyrocket at seemingly-random epochs and the accuracy would plummet as well. As such, we adjusted the learning rate from 0.001 (1×10^{-3}) to 0.0005 (5×10^{-4}), and this stabilized the loss. See Figure 6 on page 7.

As for the convolutional neural network, the original structure we used implemented 4 layers (each with a convolution and max pooling), and no Dropout layer. The accuracy for this, using 190 classes, sat at around 80% with a loss at around 1.0. Soon, we added a 5th layer following the same structure and now including a Dropout layer and the accuracy bumped up to 84% with a loss at around 0.8. See Figure 7 on page 7.

5.2 Comparison

Given the nature of the final state of our project, we attempted to compare our project with Google Tesseract, but we were unable to gain any meaningful results. Our project as it stands isn't comparable to a deployed product from which we could do any quantitative analysis. For example, Google

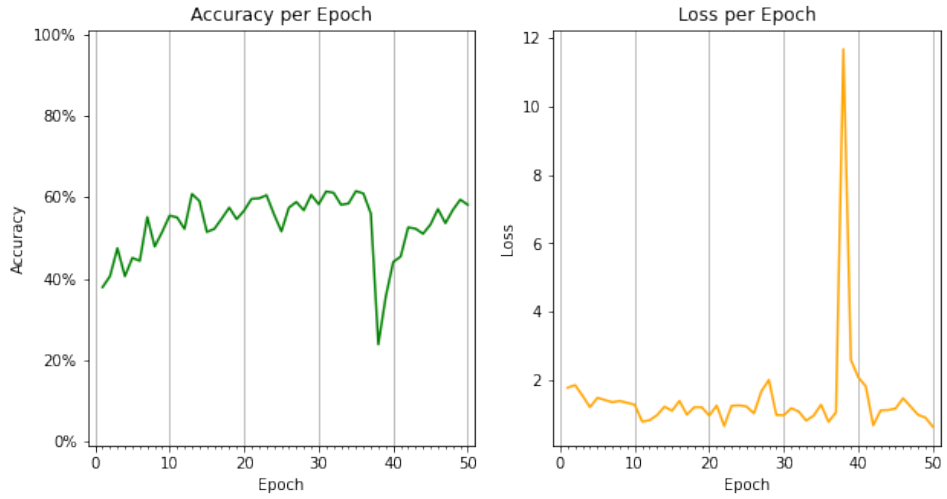

Figure 6: Feed-forward neural network loss and accuracy plots over training epochs. There is a downward spike in accuracy and an upward spike in loss at the 38th epoch. Testing accuracy plateaus at around 55%-60%. Loss remains stable ranging between 0-2.

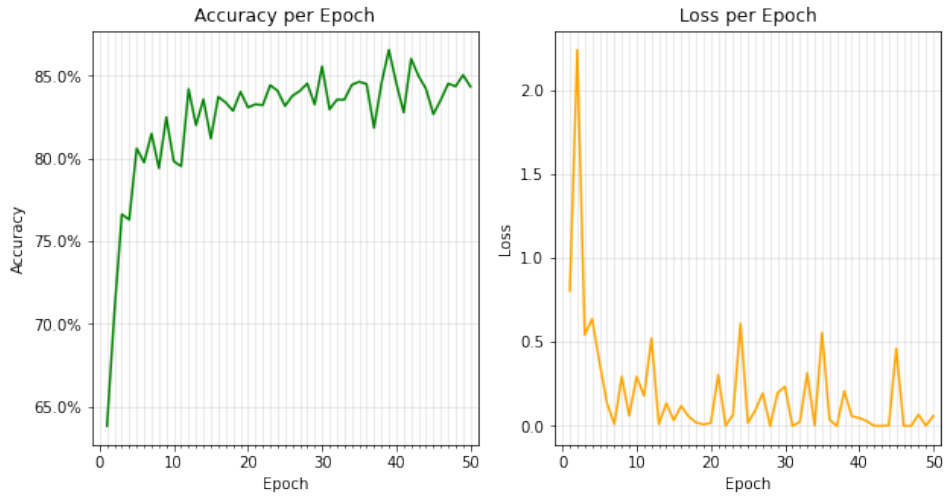

Figure 7: Convolutional neural network loss and accuracy plots over training epochs. Testing accuracy quickly reaches the 80s in percentage and loss quickly drops to roughly less than 0.5 by the 8th epoch. Training accuracy continues to increase slowly from the 20th to the 50th epoch, and loss remains relatively stable with cyclic spiking every 10-12 epochs, with spikes not exceeding 0.6.

Translate for mobile apps has a feature that allows users to hand-write Chinese characters to input into the translator, which is loosely what our current project state is based on, but once again, we were not able to find a way to do a proper analysis on performance using this. Overall, we could just base it off of our own user experience in which Google guesses the correct character almost 100% of the time, whereas our model, as stated in the model accuracy section, predicts handwritten characters correctly roughly 70%-80% of the time.

6 Conclusion and Discussion

6.1 Successes and Next Steps

We were successful in creating the baby steps towards our original goal: creating a neural network that could use live video feed to read handwritten Chinese text. The next natural steps, given that our convolutional neural network only has this success with 190 different characters, is to incrementally size-up the network to handle more classes while maintaining an acceptable accuracy. – It is worth noting that for 190 characters, an 83% accuracy is more or less acceptable given that Chinese has thousands of commonly used characters. However, if we were to implement a network to classify even just one thousand characters, we would require an accuracy of at least 90% as a lack of integrity could mean losing potential consumers of the product.

After successfully sizing up the network to handle nearly all classes, the next step would be to create a recurrent neural network that could handle a stream of characters and properly recognize that stream, for example, "你好" ("hello"). Once being able to recognize a stream of characters statically (first, two characters at a time, then any arbitrary number of characters), learning to implement this with computer vision such as the Y.O.L.O. methodology would be close to the final goal.

The final goal would be retrieving information that is recognized so that it could be implemented in something such as a translation app, such as how Google translate allows users with the mobile application to hover above foreign text and retrieve real-time translations.

6.2 Considerations

Overall, we could've implemented more models side by side and compared them all towards the end, but with our dataset being around 20 gigabytes, plus our limited computing resources (or money for services such as cloud computing), we had to work with the time available across team members. As such, we are still proud of our accomplishments thus far and are excited to learn more moving forward.

7 Acknowledgements

Our team formed around April 15th, which was the middle of week 3 of Spring quarter 2022. We spent about a week deciding on a project that we could do. Aleksa initially proposed computer vision for text recognition in English, and then extended it to Chinese. Taylor and Alec suggested ideas such as the taking a picture of text and recognizing words and/or characters from there, and also the GUI drawing-recognizing which we ultimately implemented. During this week, we searched for a dataset that met our needs such that we could proceed with our idea. By the end of week 4, we submitted our team and project idea, and by the beginning of week 5 we submitted our project proposal and initial progress report.

Keeping in mind that we had limited time and resources to work on this project, we set deadlines for stages of the project, i.e. finishing gathering and cleaning data by the end of week 6, and actively working on model building and GUI implementation during week 7. Model design, building and evaluation was handled by Aleksa. The largest roadblock in implementing the GUI was in making it properly capture the drawing. Taylor and Alec mostly collaborated on creating a working GUI that could take in a user's drawing, and that provides buttons for clearing the drawing, and saving the drawing to be forwarded to the prediction model. The overall goal was to be finished with the project by week 8 in order to allow for wiggle room for debugging and feature improvements before presenting in week 9, which we have successfully accomplished.

REFERENCES

[1] Bose, Amitrajit. "Intro to Neural Networks Using Pytorch-Handwritten Digit Recognition." Medium, Towards Data Science, 9 Mar. 2022, https://towardsdatascience.com/handwritten-digit-mnist-pytorch-977b5338e627.

[2] Liu, Cheng-Lin, and Fei Lin. "CASIA Online and Offline Chinese Handwriting Databases." 模式识别国家重点实验室, 1999, http://www.nlpr.ia.ac.cn/databases/handwriting/Online_database.html.

[3] Handwritten Chinese Character (Hanzi) Datasets. https://www.kaggle.com/pascalbliem/handwritten-chinese-character-hanzi-datasets.

[4] Nmonzon. "PDF." IPOL. URL: http://dev.ipol.im/~nmonzon/Normalization.pdf.

[5] Hashemi, M. Enlarging smaller images before inputting into convolutional neural network: zero-padding vs. interpolation. J Big Data 6, 98 (2019). https://doi.org/10.1186/s40537-019-0263-7

[6] Singh, Jaideep. "Handwritten Digit Recognition GUI App." Medium, Analytics Vidhya, 20 Apr. 2020, https://medium.com/analytics-vidhya/handwritten-digit-recognition-gui-app-46e3d7b37287.

[7] Burris, Sylvia. "Train a Custom Tesseract OCR Model as an Alternative to Google Vision for Reading Children's..." Medium, Towards Data Science, 5 Nov. 2021, https://towardsdatascience.com/train-a-custom-tesseract-ocr-model-as-an-alternative-to-google-vision-ocr-for-reading-childrens-db8ba41571c8.