Handwritten Intelligent Character Recognition:

Isolated OCR for manually filled forms

**Gourab Nath** **Ankit Sekseria** **Kedarkumar Golla**

Department of Data Science Student Student  
Praxis Business School Praxis Business School Praxis Business School  
Bangalore, India Bangalore, India Bangalore, India  
[gourab@praxis.ac.in](mailto:gourab@praxis.ac.in) [sekseriaankit0657@gmail.com](mailto:sekseriaankit0657@gmail.com) [kedarkumar.golla@praxis.ac.in](mailto:kedarkumar.golla@praxis.ac.in)

**Manish Kumar** **Rahul Aggarwal** **Varishu Pant**

Student Student Student  
Praxis Business School Praxis Business School Praxis Business School  
Bangalore, India Bangalore, India Bangalore, India  
[manishkumar.ms64@gmail.com](mailto:manishkumar.ms64@gmail.com) [aggarwalrahul31@gmail.com](mailto:aggarwalrahul31@gmail.com) [varishupant@gmail.com](mailto:varishupant@gmail.com)

*Abstract*— Technology has affected our lifestyle in numerous ways, both positive and negative. Even in the banking sector, one of the oldest professions in the world, we have seen life changing applications that have made the industry more efficient. From net banking to ewallets, we are seeing improvement in customer experience and better data protection. But everything has its ups and downs. Due to the ever-increasing risk of cyber-crimes and frauds, a lot of organizations including banks still use a lot of paperwork in their daily procedures.

A lot of this paperwork includes hand filled forms, for example- Withdrawal slips, money transfer forms, and loan application forms. These forms are characterized by small blocks or square spaces in which the required information needs to be filled. Later, executives of the bank have to do the needful and transfer this information through a computer to the bank’s database through manual typing. Our task intends to make this exhausting process of transferring information quick and easy using Optical Character Recognition. This will simplify the encumbering job so the employee and the organization both become more efficient.

Keywords—bank, college, forms, ocr, optical character recognition, handwritten ocr, intelligent character recognition, isolated ocr, hand filled, manually filled

# Introduction

As mentioned above, such standardized forms with square shaped blocks are used in various industries, one such industry being education. In educational institutes, forms are used for admission purposes as well as daily attendance. We have worked on the case of student registration forms used by Praxis Business School, Bangalore.

The form is comprised of six sections for gathering different information. Considering the top section to be the first one, here are the sections and the information gathered-

* Student ID
* Name

1. First Name
2. Middle Name (if any)
3. Last Name

* Address

1. Address for Correspondence
2. City/Town
3. District
4. State
5. Pin Code

* Parent’s Information

1. Father’s Name
2. Mobile No.
3. Mother’s Name
4. Mobile No.

* Health Information

1. Date of Birth
2. Gender
3. Blood Group

* Contact Information

1. Mobile
2. Landline
3. Email ID

The same can be seen in Fig.1

Each of these sections is a rectangle that spans almost the width of the form and each of the points or subsections where the handwriting is present is a row of consecutive blocks.

Now the information that’s been gathered can be in the form of alphabets, numbers or punctuations.

The form can of course be filled in different languages but our project works on the English alphabet.

Let’s say we consider the set of all English alphabets

such that

Now we represent the set of block or capital letters by

where is a set of length 26 and the set of all small letters by such that Here and are subsets of . Of course, . Let be the set of single digit Arabic numerals, i.e.

and be the set of 15 special characters from the English language. Here

Our project focuses on the set of characters D where

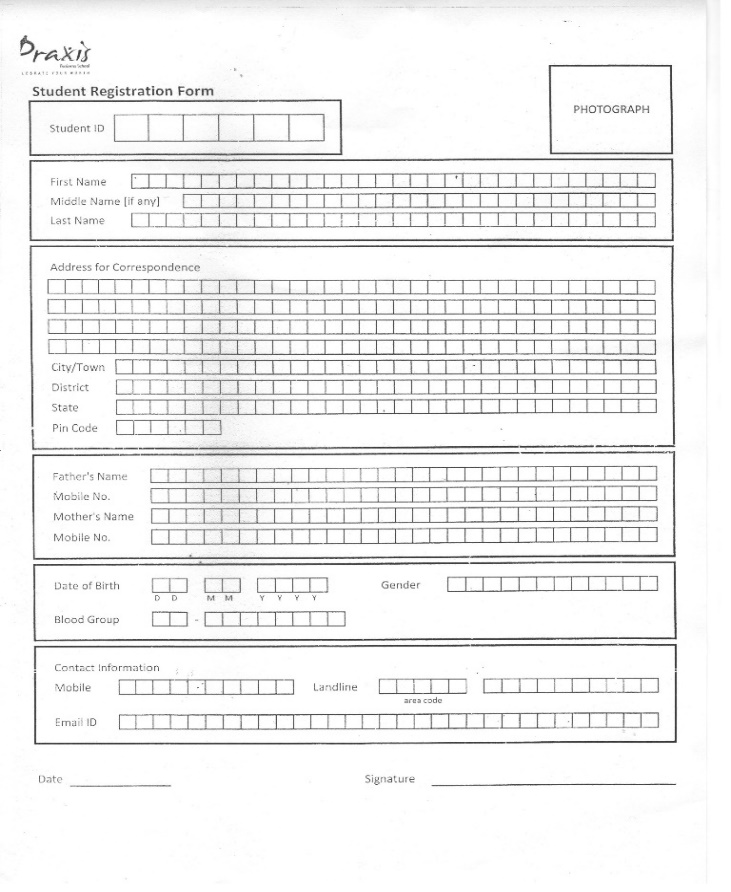


Fig 1. A sample form for which the project was made

# LITERATURE REVIEW

1. Retrieving the Image

We need a format that can be fed into the machine and that is why we take digital copies of the forms using a scanner. To help the machine read the file efficiently, various compression [1] and quantization [2] techniques can be applied.

Binarization is a form of quantization in which a greyscale image is converted into two levels, black and white. This reduces the information in the pixels from 256 shades of gray to 2. We have used the same in our work.

1. Image Preprocessing

After we retrieve the image in a digitized manner, the next step is to apply techniques such as noise removal and thresholding in order to improve the quality of the image.

[3] shows why combining multiple thresholding binarization values improves upon the output and [4] uses neuro semantic thresholding for improving the precision from classification. Some morphological operations such as dilation are combined with the thresholding for better results. Also, smoothening techniques like Gaussian blur are used for the same.

1. Character Extraction

After we are done with preprocessing, we need to actually segment out the boxes in the image in order to extract the characters from them. We call this step, character extraction. This includes contour detection using bounding rectangles and hierarchy of boxes, box cleaning which is the part where we remove noisy contours and duplicate contours and finally sorting the contours according to their pixel coordinates.

We used the Python library OpenCV [5] for the same.

This gives us the various individual boxes each containing a single character as separate images, let’s call them B-Images which we then feed into our classification model.

1. Classification

The modelling techniques that can be applied for this step are highly experimental. For example, one can use techniques like neural networks, Bayesian classifiers, decision trees etc. for this task. Our project relies on a deep learning approach for classifying the B-images into the 51 classes as mentioned above. After experimenting with different architectures, and constructing our own [6], we settled on Inception v3 [7] as it gave the best results.

# METHODOLOGY



Fig 2.

Block diagram for the implementation framework

Figure 2 shows the architecture of how our OCR application is implemented. It shows the same in 4 major steps: (1) Collecting image data, (2)processing the image to make it easier to segment out the characters, (3)extracting blocks and the handwritten characters enclosed within them and (4)training a model to distinguish between the 51 types of characters in order to be able to classify a given character into one of the 51 labels.

1. Data Collection

This step details how we collected our data and made it machine readable. Also, the problems faced and techniques used in creating our database are stated below.

1)Form Creation-

We created standardized forms that were made up of 20 rows, each containing 31 blocks. The information to be filled and the order that it was to be filled in was fixed beforehand. To be precise, the first 4 rows spanned the following phrases-

‘THE QUICK BROWN FOX JUMPS OVER LAZY DOG/”’\*&():;.,@-0123456789

THE FIVE BOXING WIZARDS JUMP QUICKLY /”’\*&():;.,@-0123456789’

This can be seen in Fig 3.

These phrases were so chosen as it encompasses the 51 classes that our OCR project targets. These 4 rows are repeated 5 times over in the whole form.

2)Batch Filling-

We took the help of our fellow Praxites to fill the forms. A total of 80 students filled out these forms in their own handwriting. The data was collected in 3 batches ,40 forms each. The first batch, let’s call it FB50 was collected from 40 students of the Jan2020 batch. The second batch or SB50 was collected from 40 students of the June2019 batch. The third batch or TB50 was collected from the same 40 students of the Jan2020 batch but after a period of 50 days. The participants filling out the form were encouraged to take their time and instructed beforehand about the format using a sample form as shown in Fig 2.

Note that the format was decided and fixed so as to make the extracted characters easy to label.

3)Digitization-

These 120 forms were then digitized into 300 dpi images using a scanner. There are guidelines and recommendations to be considered while scanning the documents. The first and foremost being the resolution of the digital copy. 300 dots per inch is considered a gold standard, although not an official one in the OCR community. It’s said to give the highest accuracy on OCR related problems.

However, when we were scanning some of the documents of FB50, the dpi was set as 200. This ended up creating problems in the character extraction stage. There is a way to overcome this problem. One can try to use scanner software to increase the resolution through interpolation. What this does is it adds extra pixels or stretches the scanned pixels which approximates the resulting image. This, however ended up making the image unclear and we lost on some of the quality of the image. This led to problems like the letter ‘B’ looking like an ‘8’. That’s why we ended up scanning the documents again, with 300dpi.

It makes sense if you understand how we define dpi. What 300dpi essentially means is that the scanner is capturing 90000 dots per square inch (300 vertical X 300 horizontal). When compared to 200dpi in which scanner captures 40000 dots per square inch, there is a massive difference in quality. In the latter case, quality drops more than 50% of the original.

Some of the other industry standards include a brightness of 50%, documents with even minor discoloration due to age to be scanned in RGB mode, etc. Another major one that can lead to inaccurate results is skewness. If the images get tilted or skewed during scanning, it can affect the later stages. That is why the straightness of the scan has huge importance.

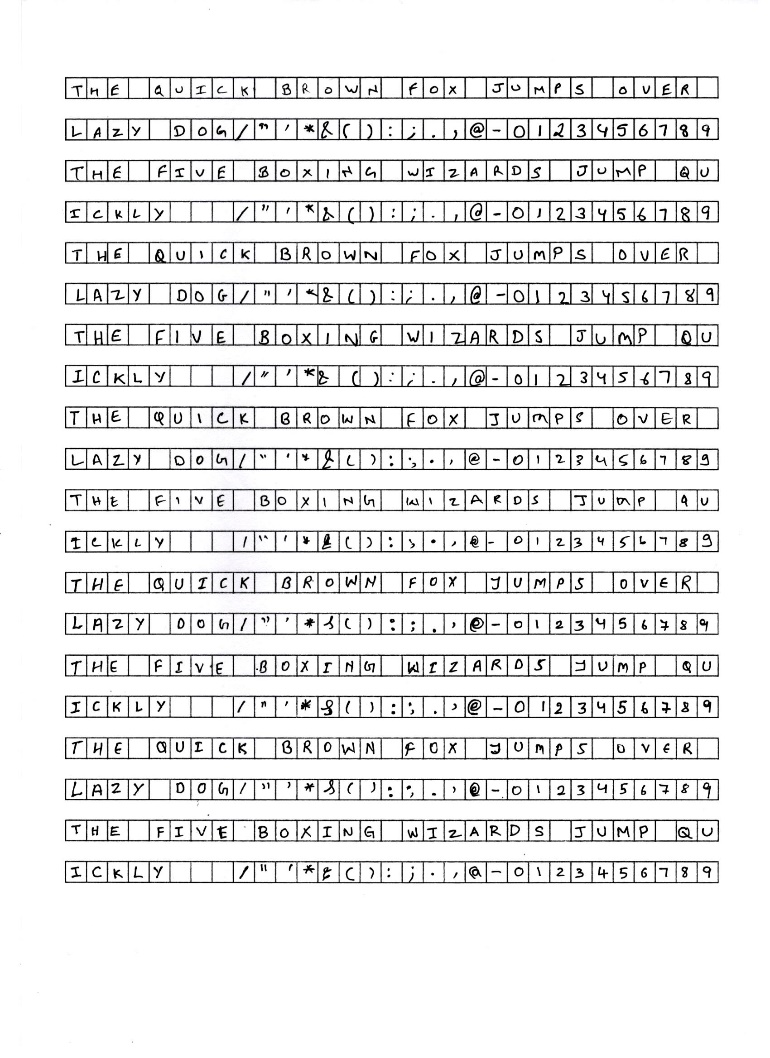


Fig.3 - A sample form used in data collection

4)Binarization-

An image is made up of pixels and each of these pixels is associated with a pixel value which represents its brightness and color. In colored or RGB images, a pixel value is a vector of 3 numbers, one for each of the components of color, red, blue and green. A grayscale image is one in which all pixels have values that correspond to only the brightness of the pixel. Essentially, a colored image can be stored as 3 separate grayscale images. For such a grayscale image, pixel values range from 0 to 255, where 0 is considered as absence of light or black and 255 is considered to be representing white. All values in between represent different shades of gray.

For text related documents, the technique of binarization is often applied to help localize the text against the background. In binarization, a threshold is chosen, let’s say 128, and each pixel having a value greater than that threshold is given a value of 255 whereas each pixel with a value less than or equal to the threshold, 128 is given a value of 0. This leads to the image converting into a black and white image or binary image. For the process of binarization, the image must be in grayscale. The process of choosing the threshold for efficient results is called thresholding and we’ll talk about that in a later section.

So, this is the reason why we read the digitized images in grayscale mode and then apply binarization.

After this step, all the images are stored in a database before passing them on to the preprocessing pipeline.

1. Image Preprocessing

Now that our database is prepared, we start with preprocessing. To do this, we take the use of inbuilt functions of the OpenCV library in Python. The steps we perform include (1) separating the text from the background, (2) increasing the relevant area including the character, and (3) removing pixel noise through blurring. Let’s look into each stage in some detail.

1) Thresholding

As stated in earlier under Binarization, images are preprocessed to binarize them using a threshold. The process of assigning pixel values using the given threshold, is called thresholding. We use this to segment out the background which is now mostly black due to binarization from the foreground, which is the handwritten character in white. In a simple thresholding operation, the same threshold value is applied to each pixel which is the same as Binarization. But in our work, that wasn’t giving us satisfactory results, which is why we turned to Adaptive Thresholding.

In this type of thresholding, the threshold for each pixel is set locally based on some small region around it instead of globally as in the case with Simple Thresholding. We used the function adaptiveThreshold from the OpenCV library. There are 2 ways to implement this thresholding, THRESH\_MEAN\_C, where the pixel gets thresholded based on the mean of neighborhood pixel values after subtracting a constant C, and THREH\_GAUSSIAN\_C, where the threshold assigned is a Gaussian weighted sum of neighborhood values minus constant C. We have used both the techniques in our work.

As the forms we worked with have 6 major sections, and multiple sub sections in each, we found that for segmenting out the outer rectangles, the Gaussian based thresholding works best while Mean based thresholding proved better for the boxes inside.

The neighborhood area that worked for us was 15 pixels around the targeted pixel and the constant C we used was 2.

2) Dilation

Dilation is one of the morphological operations used after noise removal. Using the function dilate() of the OpenCV library, we can set a kernel size that convolves the image and adds white pixels around the boundary of the character. This helps in making the character more prominent for the training algorithm so that it can learn it’s features better.

We experimented with various kernel sizes and found that dilating through asymmetric kernel sizes helped in getting the subsection boxes. For example, using combinations of (10,3) and (3,10) as kernels proved beneficial in contour detection. We hypothesize this may be the case because of our contours being square shaped. The fact that using a kernel size of (2,2) for the outer rectangles worked best, helped our assumption even more. For the outer rectangles however, dilating the image after thresholding gave better results while for the boxes, doing the reverse worked wonders.

3) Smoothening

Smoothening is another noise removal step performed in various object detection related works. In simple terms, it’s blurring the image, or sacrificing granularity in order to achieve clarity.

We did not need to use smoothening for detecting the outer rectangles, probably because of the wide area of the contours, which made the noise sparse. However due to the small and compact nature of the boxes, blurring helped in cleaning out the noisy pixels. We used the blur() function from the OpenCV library for this task. Similar to dilation, we set a kernel that slides through the image, and averages pixel values for pixels contained in the kernel. We wanted to test our earlier hypothesis about the asymmetric convolutions here and found that they indeed provide the best results. We had created 5 lists in total that were supposed to contain the total number of inner boxes in the whole form. To go through with this, we would initialize and empty list, and then do a combination of thresholding, dilation and smoothening in different sequences and with different parameters, followed by contour detection. In the end, we would take the intersection of the lists in order to remove boxes that had come multiple times. We saw through experimentation, that in those combinations that included blurring, the best results were provided when we smoothen the image first, and then go ahead with either thresholding or dilation.

1. Character Extraction

As mentioned just above, we tried different combinations of pre-processing steps followed by contour detection. This is the stage where character extraction starts. All in all, we can divide this step into 3 parts, (1) detecting the boxes, (2) removing noisy entries and (3) sorting to get the original order back.

1) Contour Detection

The findContours function in the OpenCV library defines contours as a curve joining all the continuous points along the boundary of the image having the same intensity. In our case of binarized images, the boundary of the rectangles or boxes being white has a very different intensity or pixel value than the pixels it contains. Using this function, we extract out, first the outer rectangles, then the boxes inside each one of them and finally the characters contained in those boxes. The approximation method we used was CHAIN\_APPROX\_SIMPLE which stores only the boundary coordinates of each contour instead of storing coordinates of all 4 lines that make up a contour, a rectangle or a box in our case. This was done to save memory as our pipeline was handling several images and we dreaded the terrible Out Of Memory Error.

Another useful feature of the aforementioned function is that it handles hierarchy among contours. What this essentially means is that in cases like ours, where there is a parent contour, the outer rectangle and multiple child contours, the boxes inside, it can be a difficult task to sort them all out after we find them. So, to help out Computer Vision enthusiasts like us, OpenCV numbers each of the contours and stores their relationship with their parent or child contour, if they exist in the first place. We used this feature when detecting outer rectangles by setting the contour retrieval parameter as RETR\_CCOMP. However, since we knew there isn’t any useful hierarchy inside the subsection boxes, we opted out by setting the parameter as RETR\_LIST which handles all contours as the same.

After detecting the contours, we create bounding boxes around them to initiate extraction. Here we used the Straight Bounding Rectangle through the application of the function boundingRect() from the OpenCV library.

2) Box Cleaning

Once we had the list of contours after applying RETR\_CCOMP ,we put conditions to check whether the contour is a parent and has at least one child. If that condition was satisfied by a particular contour, we checked their width. Since the outer rectangles, apart from the one containing the Student ID had equal width and also their width was significantly larger than the inner boxes, even including the rectangle for first section, we were able to successfully filter out only the outer rectangles from the contour list. But along with them, came some extra unneeded contours, which had to be cleaned. To do that, we calculated the mode of the heights of contours, and put in conditions to filter out contours with small heights. Then, there were some contours which basically represented the same rectangle but came multiple times because of the difference of a few pixels in their coordinates. To remove such duplicate contours, we put in conditions that check if the difference between any 2 contour’s boundary pixels is less than a small constant, then one of them should be removed.

After getting all the outer rectangles, we sorted them and passed each of them through a function which extracts the inner boxes. This function is the one that creates those 5 lists as mentioned several times above. These lists containing the inner boxes also need to be cleaned. So we pass all the contours in each of these lists through some conditional statements that check whether their height and width are in the acceptable region and then for removing duplicates, a similar code as mentioned above was applied. Then, these inner boxes were sorted as well.

3) Sorting

As we saw above, sorting was done in two different timepoints, one was to sort out the outer rectangles and the other was to sort the inner boxes. Now to sort the outer rectangles, we just needed to pass the cleaned contours through the sorted() function in Python giving the coordinates as the key, and because the rectangles were so far apart, there were no complications.

This was not the case for the inner boxes though. Here, the boxes were horizontally attached in a line and vertically spaced between different subsections within a section. For this task, we wrote codes which can be seen in Figure 4. This was based on the fact that for any particular subsection, take First Name for example, contours would be considered sorted when all the boxes with the same y coordinate came one after the other with a gap that was equivalent to the width of the box.

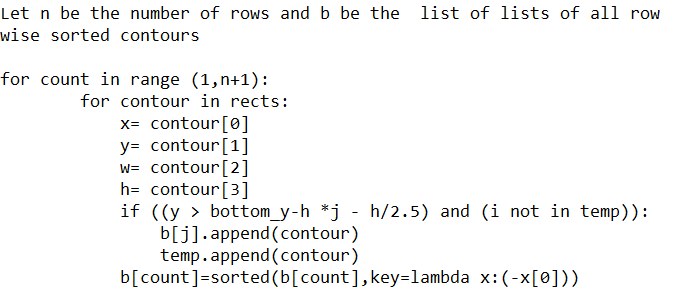


Fig. 4

Code to sort inner contours

Once we have the cleaned and sorted contours, we pass them through our classification model which predicts the character inside each contour.

1. Classification

We started by creating our own CNN architecture but the switched to pre-built architectures like Inception and LeNet due to low accuracy. Through rigorous experimentation, we finalized ResNet. Since we were extracting characters section-wise, we first trained 3 separate models to be used in separate sections. One was trained only on alphabets and white space for sections like Name, one was trained only on digits for sections like Mobile Number and lastly, one contained all the classes.

We trained ResNet with the images in several batches. As stated under Data Collection, we had 3 separate batches of forms, namely FB50, SB50 and TB50. We used these to create several more batches with a standardized naming process to prevent confusion. Here the first 2 letters of the name specify which batch of data we are referring to, that is, the first batch (FB) or the second batch (SB) and so on. The numbers following the batch name refers to the number of classes we had in the batch. For example, FB26 referred to only the alphabets from the first batch. Similar to this, various batches were designed to experiment with different models.

For an instance, FB11 was used as the training dataset and SB11 was used as the validation dataset to build a model that classifies between the Arabic numerals and a special character, that is ‘white space’. This was done in order to classify the images collected from the section- Contact Information, subsection-Mobile No.

For better accuracy, we modified the architecture by adding dense layers to the fully connected layers at the end. This was done to ease the architecture into condensing the output to 51 probabilities, since we used SoftMax as the activation function.

We also used augmentation techniques like shear, zoom, and rotation to make the model versatile for different handwritings. Since the FB and TB batches were collected from the same students, we expected a higher validation accuracy because of the model being trained on the characters written by the same writers. We found the same, when the model was trained without augmentation but that problem was solved when images were augmented before fitting the model.

We trained the model from scratch while keeping the input shape as (75,75,1). Global Average Pooling was performed before the dense layers were added near the end of the architecture. Early stopping criteria with patience set as 8 was used during training along with a dynamic learning rate that was reduced by a factor of 0.1 on encountering plateaus.

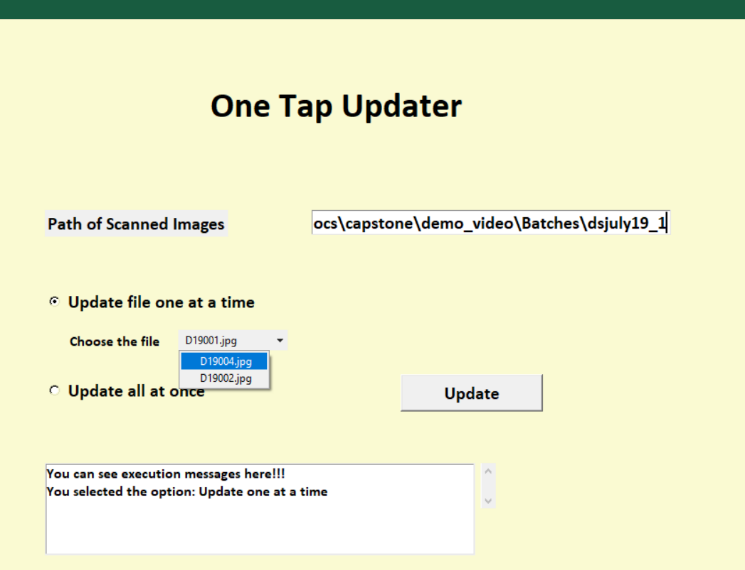


Fig. 6

Screenshot of GUI and Format of Output

# FINDINGS

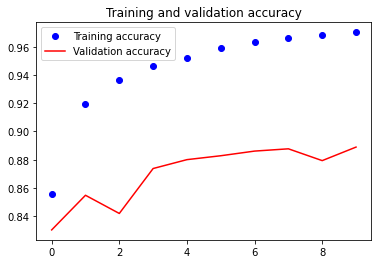
In this paper, we have conducted experiments on hand filled forms collected from our friends at Praxis. These forms were scanned, and the characters were extracted, then a classification model was trained on them. The final model was trained on 80 such forms as seen in Figure 3 with the inclusion of NIST [8] data to enhance the training of alphabets and digits and validated on 40 forms. For testing, we used forms such as seen in Figure 1.

We used TkInter, a python GUI toolkit to create an application that takes scanned images of forms as input [Fig 1] and outputs an excel sheet with the information in the form. Fig.6 shows the same. The row shown acts as headers and corresponding information is stored below. Each form corresponds to one row in the sheet. Also, the option of batch update was available. By clicking on the option, Update all at once, the user can update all forms in the folder whose path is provided.

Then we used PyInstaller to convert our script into an executable file. Here we faced a problem: time taken for conversion was huge and not scalable for large amounts of files during batch update. To overcome this, we took 2 steps:

1. Removed 4 classes that were not getting encountered even once, namely "(quotation marks), '(apostrophe), ;(semi-colon) and \*(asterisk).
2. Instead of using multiple models, we used only one, trained on all 47 classes.

We trained ResNet for 10 epochs using Keras and achieved an accuracy of 89% during inference as shown in Figure 7. The errors we were getting were predictable, for example, the model confused 0 with O. So, we coded some custom rules to overcome this and improved the accuracy further.



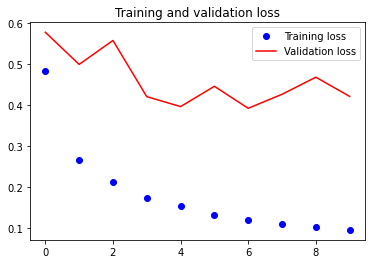


Fig. 7

Loss and Accuracy Curves

The application seen above tracks the files already updated in order to prevent mishaps like accidental duplicate updates. This is based on the assumption that each form is saved with a unique name. For example, in our example, each form corresponds to a student registering in Praxis Business School and each student has been assigned a unique identifier. The folder used in the GUI screenshot [Fi. 6] has 3 files, namely D19001, D19002, D19004. This is done intentionally so that each scanned image is named using the unique ID of the student who filled the form. This satisfies the aforementioned assumption and hence the application works fine.

This project, although currently specialized for Praxis student registration forms has the potential to be scaled up and be generalized for all manually filled forms with isolated blocks. We encourage you to take it to the next level!

##### References

1. Bhammar, M.B., & Mehta, K.A, 2012, Survey of various image compression techniques. International Journal on Darshan Institute of Engineering Research & Emerging Technologies, 1(1), 85-90
2. [6] Bhavani, S., & Thanushkodi, K, 2010, A Survey On Coding Algorithms In Medical Image Compression. International Journal on Computer Science and Engineering, 2(5), 1429-1434.
3. [9] Lund, W.B., Kennard, D.J., & Ringger, E.K. (2013). Combining Multiple Thresholding Binarization Values to Improve OCR Output presented in Document Recognition and Retrieval XX Conference 2013, California, USA, 2013. USA: SPIE
4. Lazaro, J., Martín, J.L, Arias, J., Astarloa, A., & Cuadrado, C, 2010, Neuro semantic thresholding using OCR software for high precision OCR applications. Image and Vision Computing, 28(4), 571-578.
5. @article{opencv\_library,

author = {Bradski, G.},

citeulike-article-id = {2236121},

journal = {Dr. Dobb's Journal of Software Tools},

keywords = {bibtex-import},

posted-at = {2008-01-15 19:21:54},

priority = {4},

title = {{The OpenCV Library}},

year = {2000}

}

1. Ciresan, D.C., Meier, U., Gambardella, L.M., & Schmidhuber, J, 2011, Convolutional neural network committees for handwritten character classification presented in International Conference on Document Analysis and Recognition, Beijing, China, 2011. USA: IEEE.
2. Rethinking the Inception Architecture for Computer Vision By Christian Szegedy and Vincent Vanhoucke and Sergey Ioffe and Jonathon Shlens and Zbigniew Wojna, 2015.
3. Shen, V.K., Siderius, D.W., Krekelberg, W.P., and Hatch, H.W., Eds., NIST Standard Reference Simulation Website, NIST Standard Reference Database Number 173, National Institute of Standards and Technology, Gaithersburg MD, 20899, <http://doi.org/10.18434/T4M88Q>