Hebrew OCR Machine Learning Project

By Cody Sensenig and Seth Howell
CS 422 Machine Learning
May 05, 2021

Our Need for Data and Source of Inspiration

The MNIST dataset has been the foundation of many optical character recognition (OCR) projects since its creation. This dataset of handwritten digits consists of 60,000 images in the training set and 10,000 in the test set. It is a subset of the larger NIST dataset. However, since Seth was already working towards building a Hebrew literacy app, we wanted to do the same thing with Hebrew handwriting. We used the description of the data preprocessing for MNIST to get a sense for what we wanted. Then we began the search for a Hebrew character dataset [1]. Only one option existed.

Our work on this project was inspired by a paper written for the International Conference on Frontiers in Handwriting Recognition. In this paper Irina Rabaev etal presents the HHD or Hebrew Handwriting dataset. This dataset contains 1,000 images of handwritten Hebrew text. It is the only one of its kind in existence today [4]. However, this dataset uses cursive handwriting whereas we wanted to be able to recognize printed handwriting. This necessitated the creation of our own dataset.

Our Collection of Data

We modeled the data collection process off of the NIST dataset where they had students fill out a sheet of paper [5]. This paper was arranged in such a way that the characters could then be easily extracted and converted into a binary format. This could then be pre-processed and fed into a python algorithm.

To accomplish this we used Microsoft Excel to create an input sheet that was composed of 30x30 squares. Each sheet has 28 unique characters–each hebrew letter + sofit forms–and five boxes for each one (5 columns). This means that each sheet produces a total of 140 individual data points (characters). The sheet was arranged with an example letter at the beginning of each line making it easy for people to copy the letters over. Later on we had to mess with the density and color of the lines so as to make sure they disappeared when passed into the algorithm.

We printed around 30 of these sheets and handed them out to the student body to get a good sample of data. Upon receiving the sheets back, we used the printer to scan them and send them to our emails. Then we opened them in GIMP, cropped them and fixed the perspective. This

allowed us to pass a clean cut image into the pre-processing algorithm. We ended up with 20 photos and 2800 data points.

Preprocessing

We based our preprocessing off work done by Mikalai Chaly [2]. The challenge before us was to quickly label and clean enough character images to reach an acceptable accuracy for training and testing a model. Chaly's Python notebook takes as input an image with equally spaced handwritten characters such that it can be chopped into square images of a specified size for each character. His process includes the following steps:

1. Change the input image's color to black-and-white in order to reduce from 16,777,216 possible shades to 256.
2. Then the image is cut into *x* square images of a specified size *S*S* pixels.
3. Each of the *x* images are centered and cropped to a specified *S*S* pixels.
4. The output *x* images are then converted to a numpy binary format and saved locally.

For our data, we used Chaly's code and our own functions to produce the following procedure:

1. Specify an input directory that contains all of our cropped 150*840 pixel png images and a dictionary called *ALEFBET* with the name of each Hebrew character, including sofit forms.
2. In *imgs_to_bw(imgs)*, convert each input image in *imgs* to b&w and increase the contrast in order to make the letter pixels stand out more.
3. In *img_to_alefbet(img)*, cut an input image, *img*, into 140 unique letter images (28 letters * 5 occurrences per letter), 30*30 pixels each, and return an *alefbet* dictionary where each key is a Hebrew character *k* paired with a list of *k*'s five 30*30 image instances as its value.
4. In *resize_chars(alefbet)*, for each of the letter images, *char*, in *alefbet*: invert *char* to set all background pixels to value 0 (black), call *crop_center()* to remove 2 pixels from each side of *char* (this removes any residual lines from the table that was on the input form), call *getbbox()* to get only the area containing the handwritten letter and then crop *char* to that area, adding padding around the centered *char* to make it size 28*28 pixels, and finally insert it back into its location in *alefbet* and return the update *alefbet.*
5. In *make_dirs()*, create a local output folder with 28 subfolders, each named after a letter in *ALEFBET*.
6. In *save_data()*, enumerate over a list of *alefbet* dictionaries and save each letter image to its corresponding output folder, resulting in all of our pre-processed letters being labeled by the name of the directory in which they reside.

7. In *process_data()*, pass all of our input images through the three functions listed above (2-4) and call *make_dirs()* and *save_data()* to end up with our cleaned, labeled images.
8. Finally, we use Chaly's code to convert all of our labeled images to a numpy array called *binary_samples*, which contains the images in binary format, and a numpy array called *classes*, which contains their class labels in binary format. We then save these locally as 'digits_x_test.npy' and 'digits_y_test.npy'

Setting Up Our Neural Network

For our model we used the Keras Sequential neural network from MNIST Fashion tutorial on TensorFlow since it was also working with 28*28 pixel images [3]. The tutorial used a flattened 28*28 (784) array of nodes as the initial layer, followed by a 128 node dense layer with ReLU activation and a 10 node dense output layer (mapping to the 10 classes of clothing). We modified it to have an extra layer of 512 nodes to optimize the accuracy and changed the output layer size to 28 nodes to match our 28 Hebrew letter classes:

```
model = tf.keras.Sequential([
    tf.keras.layers.Flatten(input_shape=(28, 28)),
    tf.keras.layers.Dense(512, activation='relu'),
    tf.keras.layers.Dense(128, activation='relu'),
    tf.keras.layers.Dense(28)
])
```

By adding the extra layer we were able to improve the accuracy by about 2-3 percent.

Training

With our data preprocessed and labeled, we had a total of 3,780 images converted to their binary representation in our dataset (135 for each Hebrew letter). We used Sklearn's *train_test_split()* function to randomize our data and split it into training data (70%) and testing data (30%). We divided all the binary values by 255 to arrive at color values between 0 and 1 (normalization). We then compiled our neural network using Adam as our optimizer and accuracy as our metrics. We used the Keras *fit()* function to train our training data, finding that about 20 epochs produced the best accuracy:

```
model.compile(optimizer='adam',
              loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
              metrics=['accuracy'])

model.fit(train_images, train_labels, epochs=20)
```

Testing and Validation

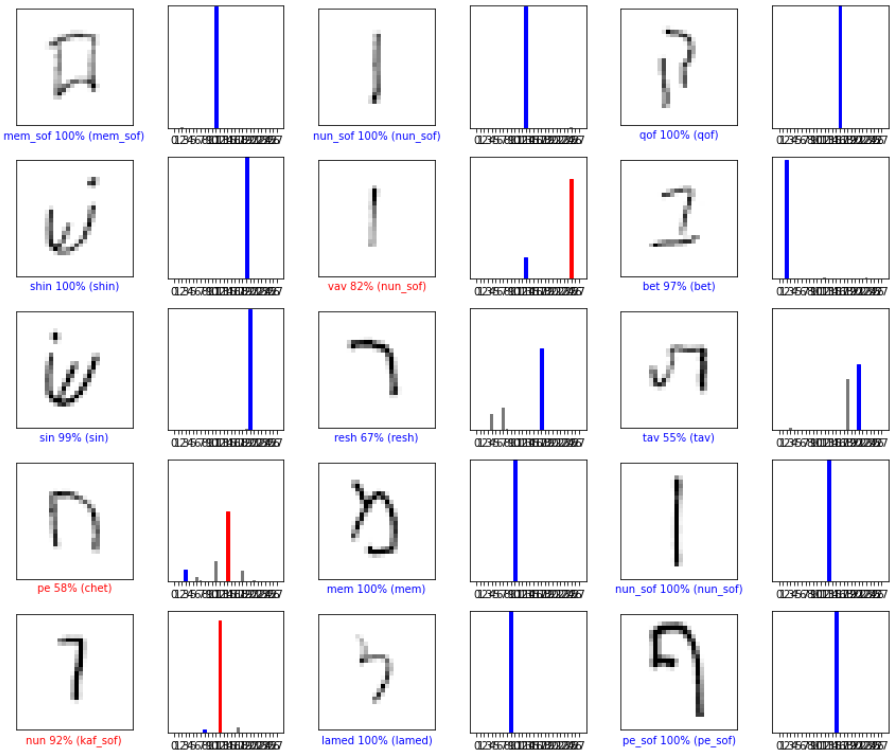
We then evaluated our model using our testing data and created a probability model that would assign a numerical value between 0 and 1 for each class label for a given image, the sum of all values equalling 1. In this way, the largest value from the set of 28 values would represent our prediction for a given image. Using this metric, we we able to consistently get an accuracy between 78 - 82 percent:

```
test_loss, test_acc = model.evaluate(test_images,  test_labels, verbose=2)
probability_model = tf.keras.Sequential([model, tf.keras.layers.Softmax()])
predictions = probability_model.predict(test_images)

print('\nTest accuracy:', test_acc)

# -> 36/36 - 0s - loss: 0.9957 - accuracy: 0.8122
```

We were also able to plot our predictive model in order to visualize its accuracy:



As one could guess, our model had the most difficulty with Hebrew letters that look very similar (e.g., vav vs. nun sofit).

Discussion On Future Work

In combination with the Hebrew literacy app that Seth is designing for his capstone, there is a lot of potential for expanding this project to other areas in the future.

One feature which would save Hebrew students time and energy would be a Hebrew handwriting autograder. It would take the input of the students handwriting through their smartphone camera and provide feedback on sentence structure, vowels and much more. In addition, one could also use it to correct one's handwriting as it could recognize the disparity between what is written and what it is supposed to look like.

It could also be expanded to help Sattler students in their first year of Hebrew. You could build Schumann's homework into the application. Then when the student scans it, there would be instant feedback and it would turn it into Populi.

References

[1] “The MNIST Database of Handwritten Digits.”
http://yann.lecun.com/exdb/mnist/.
[2] “DIY MNIST Dataset” by Mikalai Chaly.
https://www.kaggle.com/mikalaichaly/diy-mnist-dataset.
[3] “Basic classification: Classify images of clothing” by TensorFlow.
https://www.tensorflow.org/tutorials/keras/classification.
[4] “The HHD Dataset”
https://www.researchgate.net/publication/343880780_The_HHD_Dataset
[5] “The NIST Dataset”
https://www.nist.gov/srd/nist-special-database-19