Character Recognition in Natural Scenes with Chars74K and CNNs

Problem Statement

In this project, I will tackle the problem of character recognition in natural scenes - that is, images obtained through photographs of objects in the real world such as street signs, business signs, and other sources of text, rather than images of computer fonts or similarly normalized text. These “natural” images display a far greater degree of variability in text styles than computer fonts, so they pose a more significant challenge in the realm of computer vision; classical OCR methods do not perform nearly as well on natural images as they do on computer-generated text.

The applications of character recognition in these contexts are very broad. Some of the tasks enabled by this technology are automatic driving, testing the robustness of CAPTCHA anti-bot security, and developing tools to assist the blind or visually impaired.

The dataset I will use for this project is the Chars74K dataset, which was composed by Teo de Campos, et al, alongside a publication describing the team’s methods and results. The full dataset includes sets of both English alphanumeric characters and Kannada characters, as well as a number of handwritten and computer font characters. I intend to focus on the English alphanumeric characters set, which contains 7705 “good” images. An additional 4798 “bad” images are included in a separate folder, but these were not included in the published study due to their excessive occlusion, noise, or low resolution, so I likewise will use only the “good” part of the set.

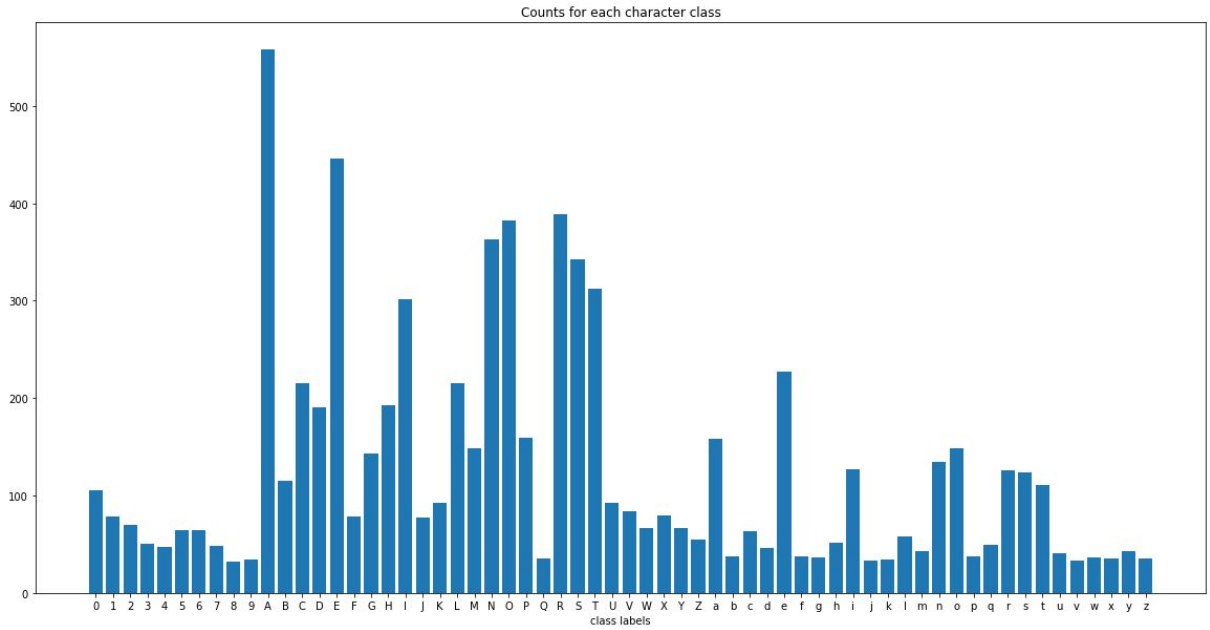
I intend to use a convolutional neural network (CNN) for this project. CNNs are a powerful tool for image classification problems, so they are well suited to the project I have laid out here. Upon completion of the project, I will have a final report, a slide deck, and project code to deliver.

Dataset Description

As I mentioned above, the segment of the Chars74K dataset that I intend to use contains 7705 images. These images may be divided into 62 classes: 10 for each digit including zero, 26 for each uppercase character, and 26 for each lowercase character. Obtaining the dataset involved only downloading it from the Chars74K web page. The dataset is mostly well-maintained; however, 5 images, all from the lowercase-q subset, had to be removed from the sample set due to file type incompatibility. Since only a very small number of images were removed, I do not expect this to meaningfully impact the model's results.

Cleaning and Exploration

After obtaining the data, I performed a number of preprocessing and data exploration steps. First, I counted the number of samples in each class, which I have plotted below.

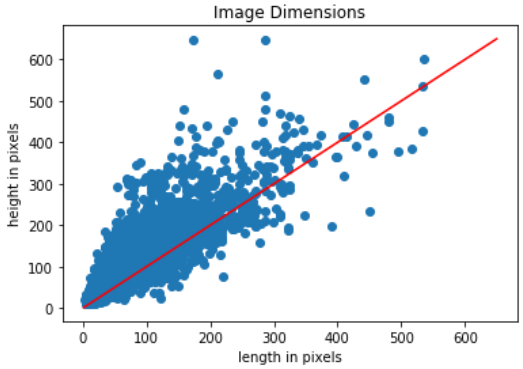


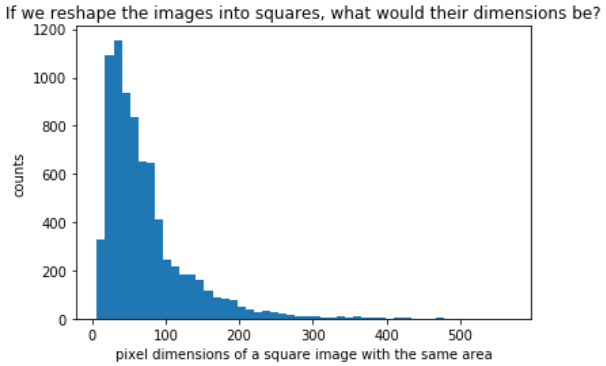
It is immediately apparent that there is significant class imbalance in this dataset. There are a few ways I could choose to deal with this. Teo de Campos recommends testing and training on only 15 samples from each class, though I have seen others use data augmentation or compute class weights instead. I initially considered it important to be

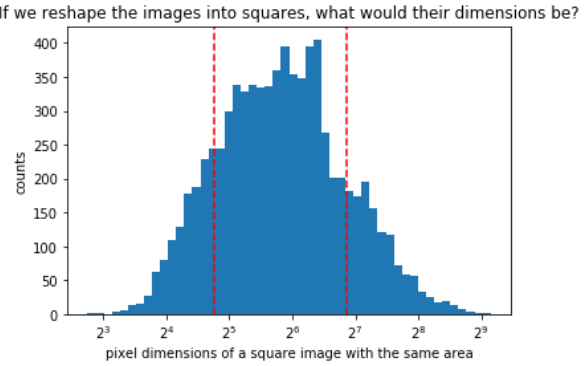
able to draw direct comparisons between my model and the original Chars74K paper, but it seems that enough work has been done using this dataset during its lifetime that this is not necessarily the case. With that in mind, I opt to compute class weights since this is the simplest technique to implement.

Some other statistics to note about the class distribution are its maximum value, 558 for capital A; its minimum value, 32 for number 8; the corresponding range, 526; and the mean and median values, 124.19 and 78. For those classes with a low number of samples, the fraction used to train the model approaches 50%, though this fraction is significantly lower for classes that have more samples.

In order to feed the images to the model, they will all need to have the same dimensions, and the original images have a wide range of dimensions. In order to determine appropriate dimensions for the images, I create a scatter plot of the length and width of the images and two histograms derived from the images' total areas. For the histograms, the actual values plotted are the square roots of the areas, which tells me what length in pixels to use; the first histogram is plotted on a linear scale and the second on a logarithmic scale.







The scatter plot offers some clues that a majority of the data are clustered toward the smaller dimensions, which is confirmed by the first histogram. Although most of the images appear to be taller than they are wide, I settle on using square dimensions for the sake of model simplicity since this seems to be a reasonable approximation of the data. Furthermore, data augmentation used to increase the sample size of image sets often involves stretching and resizing images in order to create distinct, yet still recognizable, variations of a class, so I believe that any transformations induced by this uniform resizing strategy are more likely than not to yield better predictions.

One technique that caught my interest when researching convolutional neural networks was progressive resizing. I will save a more technical explanation for when I get to actually building the model; for now, the thing to know is that it will involve successively doubling the images' dimensions. With this in mind, I select a range of pixel dimensions that contains the largest number of images, which, due to the logarithmic plot's approximately normal distribution, happens to occur at the central peak of that plot. Based on this data, I decided to use pixel dimensions of 32x32, 64x64, and 128x128.

Overview of Model Techniques

A technique central to my strategy in this project is progressive resizing, which is an application of transfer learning. Transfer learning involves using the layers and weights from a pre-trained model as the basis for a new model. When applied to progressive resizing, this means that each iteration or generation of the model will use layers from the previous generation, plus any layers needed to take the resized images as input.

Using the dimensions I selected during exploration (32x32, 64x64, and 128x128), I created a three-generation model where each successive generation takes the next largest input size . Each generation will use a training set from different random selections of the full dataset.

Building and Training the Model

Setting on an architecture for the layers of the convolutional neural network took time and experimentation. I tested several variations in the kernel size, initial number of filters, dropout percentage, frequency of dropout layers, and the number of

convolutional layers to use between each max pooling operation. The architecture I settled on for the initial model was ultimately the one printed below.

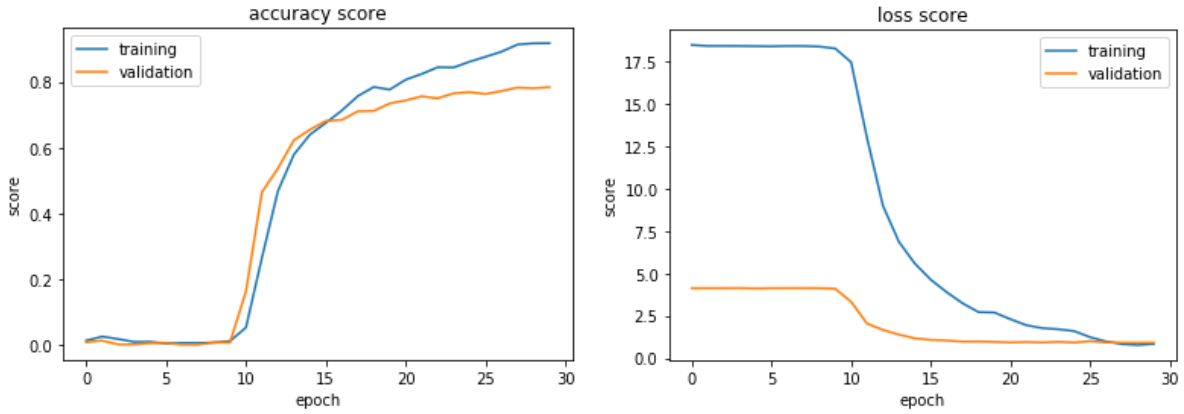
```
# CNN layers, 32x32

layers = [
    Conv2D(128, kernel_size=3, padding='same', activation='relu', input_shape=(size, size, 3)),
    MaxPooling2D(padding='same'),
    Dropout(0.2),

    Conv2D(256, kernel_size=3, padding='same', activation='relu'),
    MaxPooling2D(padding='same'),
    Dropout(0.2),

    Conv2D(512, kernel_size=3, padding='same', activation='relu'),
    MaxPooling2D(padding='same'),
    Dropout(0.2),

    Flatten(),
    Dense(2048, activation='relu'),
    Dropout(0.2),

    Dense(62),
    Activation('softmax')]

model_1 = Sequential()
for layer in layers:
    model_1.add(layer)
```

During the convolution phase, the layers are divided into a repeating pattern of three layers: one convolutional layer with the ReLU activation function, one max pooling layer, and one dropout later. After the last repetition, the input is flattened into a shape that can be fed into a dense layer. One more instance of dropout occurs before the final dense layer, which instead uses the softmax activation function to output the per class probabilities for each input.

Each generation of the model uses two callbacks: one to exponentially reduce the learning rate if no significant improvement in the validation loss occurs after a number of epochs, and a second to save the best version of the model to use as the next generation's prior. The model is compiled using the Adam optimizer, categorical crossentropy as the loss function, and accuracy as an additional metric to evaluate during training and validation. Validation is performed using 25% of the test set.

```
# build and run model, 32x32

reduce_lr = ReduceLROnPlateau(monitor='val_loss', factor=0.1, patience=5, verbose=1, min_delta=0.01)
mc = ModelCheckpoint('best_model_32.h5', monitor='val_loss', mode='min', save_best_only=True)

model_1.compile('adam', 'categorical_crossentropy', ['accuracy'])

model_1.fit(X_train, y_train,
          epochs=30, batch_size=64,
          validation_split=0.25,
          class_weight=class_weights,
          callbacks=[reduce_lr, mc])

plot_model_history(model_1.history.history)
```

After fitting the model, I plot the accuracy and validation loss scores for both the training and validation sets.

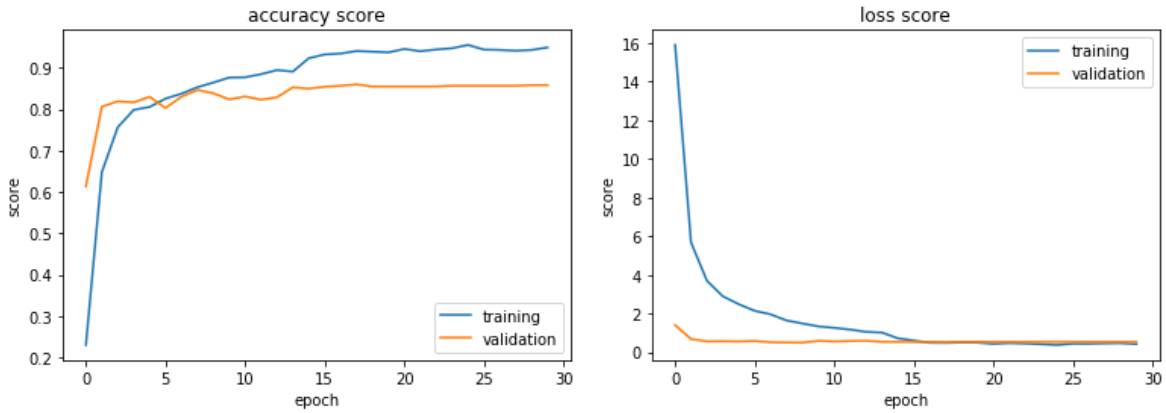


The training loss is initially much higher than the validation loss, which I thought was unusual. As best as I can tell, this is probably a result of the training model also being subject to regularization loss via the dropout layers. The gap between the two loss scores shrinks to nothing after the model begins its climb down the gradient descent. The 29th epoch displays the best validation loss, and the validation accuracy at that stage is about 78.05%. This is the version of the model that gets saved for future use.

The second generation of the model begins with a new set of input layers to accommodate data that has now doubled in size before the layers from the previous model, along with their corresponding weights, are added.

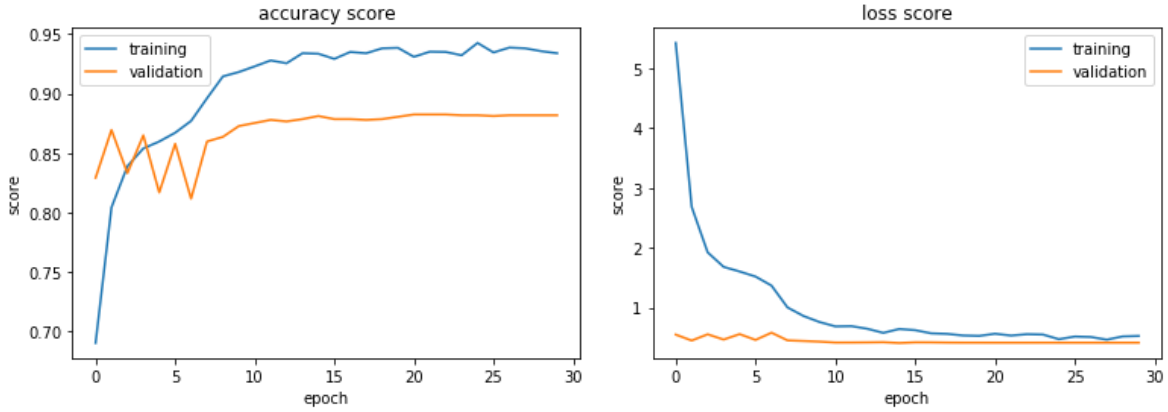
```
# CNN layers, 64x64

model_2 = Sequential()

model_2.add(Conv2D(64, kernel_size=3, padding='same', activation='relu', input_shape=(size, size, 3)))
model_2.add(MaxPooling2D(padding='same'))
model_2.add(Dropout(0.2))

model_2.add(Conv2D(128, kernel_size=3, padding='same', activation='relu'))

prior = load_model('best_model_32.h5')
for layer in prior.layers[1:]:
    model_2.add(layer)
```

The same callbacks and compiler parameters are used for the second generation as the first, and the loss and accuracy scores are plotted in the same manner as before.



The new model shows some improvement on the prior model, with the saved version scoring 0.4954 on the validation loss and 83.83% on the validation accuracy. Perhaps as a result of inheriting weights from the prior model, it takes far fewer epochs for this generation to begin to converge on a maximum accuracy and minimum loss.

Next, I train, validate, and plot metrics for the third generation.



Of all three generations, this latest one appears to benefit the most from learning rate reductions, seeing how the metrics were slowly trending in the wrong directions for the first few epochs. By the time this model finishes training, it achieves the best metric scores so far, with a validation loss of 0.4096 and a validation accuracy of 88.12%. The final saved version of the model is the one that will be used to perform evaluation in the following section.

Evaluation

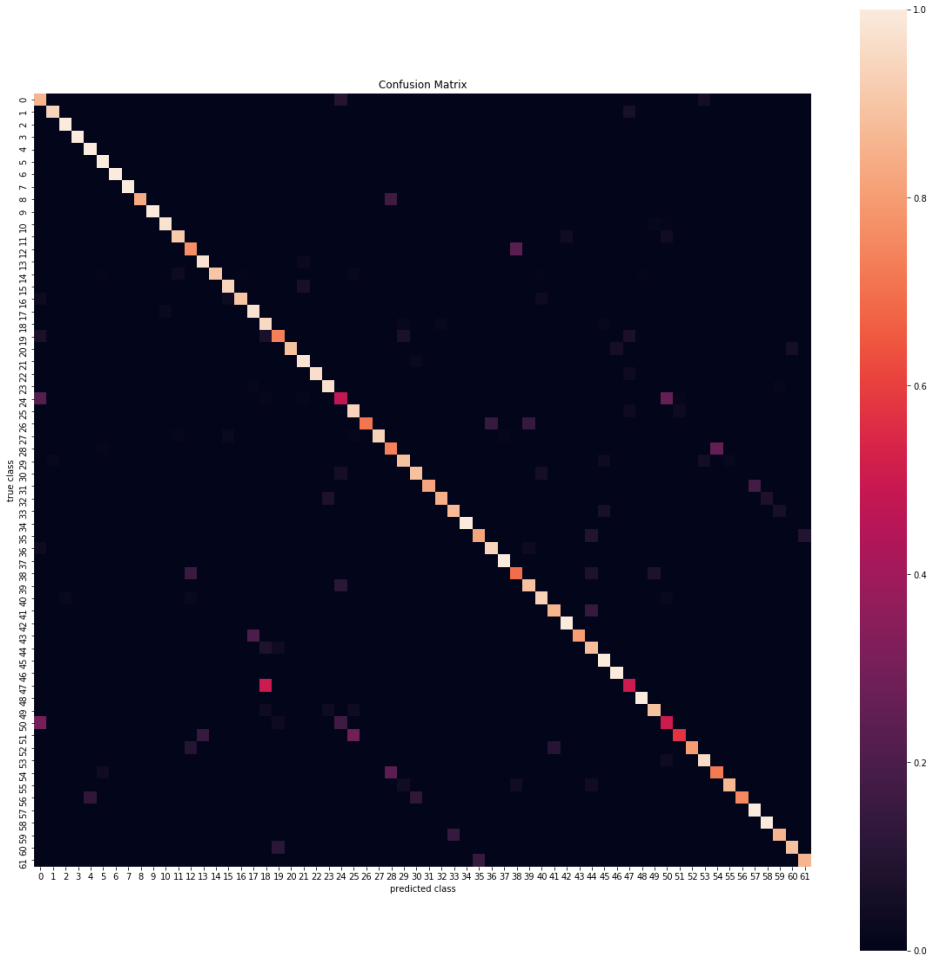
Upon evaluating the savel model on the test set, I obtain the following results:

```
13/13 [==============================] - 9s 681ms/step - loss: 0.4082 - accuracy: 0.8740
```

The loss score of 0.4082 and the accuracy score of 87.40% are within a reasonable margin of error of the validation scores for the test set, and based on other work that has been done on this dataset, I believe these final results indicate strong model performance.

Although most of the work I found on this dataset uses accuracy as the primary evaluation metric, I believe it is also valuable that we examine the precision, recall, and f_1 scores of the model. To do this, I generate a confusion matrix of the data. The confusion matrix shown in this paper is a simple heatmap that effectively conveys general impressions of each class's performance; the sheer number of classes unfortunately makes it difficult to display a confusion matrix with explicit numeric data in this format.

A larger, more detailed confusion matrix can be found in the Jupyter Notebook that accompanies this report.



The most noticeable trend I see in this matrix, besides the prominent main diagonal, is the presence of two other diagonals that each represent a set of misclassified samples. Inspecting these diagonals reveals that both of them are the result of confusing uppercase and lowercase letters for each other. Given the number of letters that maintain the same shape in each of these forms, I find the model's confusion in this area to be understandable.

Using the data from this confusion matrix, I compute a precision score of 87.06%, a recall score of 87.96%, and an f_1 score of 87.51%; on average, it seems the model performs well enough on individual classes in spite of the unbalanced nature of the data. However, the performance across classes is clearly not uniform, so it would be valuable to examine and diagnose the lowest-performing classes, which may offer insight into ways to further improve the model.

Failure Analysis and Ideas for Improvement

Below is a dataframe containing statistics for the ten lowest performing classes.

	true_positives	false_positives	false_negatives	precision	recall	f1
o	15.0	24.0	15.0	0.384615	0.500000	0.442308
l	6.0	4.0	6.0	0.600000	0.500000	0.550000
c	9.0	11.0	4.0	0.450000	0.692308	0.571154
s	18.0	17.0	7.0	0.514286	0.720000	0.617143
p	4.0	2.0	3.0	0.666667	0.571429	0.619048
0	18.0	29.0	3.0	0.382979	0.857143	0.620061
O	36.0	9.0	40.0	0.800000	0.473684	0.636842
J	11.0	3.0	4.0	0.785714	0.733333	0.759524
x	6.0	2.0	1.0	0.750000	0.857143	0.803571
S	50.0	7.0	18.0	0.877193	0.735294	0.806244

Of the ten classes with the lowest f_1 scores, o, c, s, p, O, x, and S appear on both of the diagonals associated with uppercase-lowercase confusion. I would say that all of these letters may reasonably be confused in this way even by humans.

Of course, when actually reading text rather than identifying isolated characters, humans are far less likely to confuse an uppercase letter for its lowercase form. One possible explanation for this is that we do not simply identify letters isolated from their

context the way computer vision software tends to segment letters from each other; we tend to read letters as entire words and distinguish uppercase and lowercase letters based on their relative size to other letters. If a character recognition model could be made to know which characters came from which pre-segmented image, as well as the sizes and tentative predictions of those images, perhaps a model would be able to more accurately identify the case of these confusing letters.

The classes 0 and l are similarly easy to diagnose. Class 0 frequently gets confused with both forms of the letter o, which is understandable given their shared elliptical shape. Additionally, fully half of the l (lowercase L) samples get confused for I (uppercase i). This second problem is especially understandable given that these two characters are nearly indistinguishable in certain computer fonts, including the Arial font I have been using to write these reports - I had to mark these two classes with their opposite-case form to properly distinguish them! Presumably, class I (capital i) did not struggle as much because enough samples featured serifs at the top and bottom of the character to make the appropriate class more obvious to the model.

As in the uppercase-lowercase problem, human readers usually distinguish between these characters based on their broader context. However, the kind of information used to make these distinctions is a bit more abstract than incorporating the size of characters from the same scene. For example, I and l are nearly impossible to distinguish without the appropriate context, but I imagine that the reader does not fail to identify the first-person singular pronoun “I” when I use it in this sentence. What I am suggesting is that natural language processing techniques may be necessary to make this kind of distinction.

Class J does not have as tidy an explanation for its failure cases. There are 4 samples with the J label that get misclassified and 3 samples that get erroneously classified as Js themselves, but each of these exhibits only one misclassified sample. My first thought was that perhaps this class was undersampled, so I retrieved its count and found it to be 77.

The median class count for this dataset is 78, and the train-test splits were all stratified according to class labels, so it does not appear that undersampling is the primary culprit. A second thought that occurred to me is that J has a number of different stylistic variations; in particular, a cursive J may look very different from a typeface J. It may be that the misclassified samples contained one of these variations. If this is the case, then the most straightforward solution would still be to use a larger number of samples, possibly implementing data augmentation.