CNN for Japanese Hiragana Classification

Samuel (Chase) Brower

May 2022

Contents

1 Abstract

This project is a variation of MNIST hand-written digit recognition. Instead of ten digits, I will be trying to classify the 46 Japanese hiragana (pictured in **Figure 1**). The project will involve the use of data splitting, augmentation, regularization, and pre-trained models to produce the best model for predicting data. The central architecture is the convolutional neural network, which is the foremost technology for image classification in deep learning.

1.1 Why Hiragana?

The Japanese hiragana add two novelties to MNIST that I would like to explore.

Figure 1: The Japanese Hiragana.

- First, there are far more–about 5 times as many–characters to recognize. Theoretically, this will make it so that the same accuracy is harder to achieve. This is not certain, though, since we could imagine data with many categories that would be easy for a neural network to accurately classify.
- Second, there are many embedded visual features that exist within multiple hiragana (see **Figure 2**). It will be interesting to see whether the network is capable of finding these features and classifying hiragana according to the combination of these features.

Figure 2: These two characters contain a similar feature, highlighted in red.

2 Data Preparation

2.1 Data Gathering

I wrote all of the hiragana for the data-set myself. As such it might be more fair to consider my project one to classify my own handwriting, rather than the Japanese hiragana in any form. I wanted to automate the process of saving and organizing the digits, so I wrote a program in processing (based on java). The program (see **Figure 3**) requests the user to write a particular hiragana, and grants a few features to undo mistakes and submit the final result. When the result is submitted, the program stores the drawn character in its respective folder. Finally, the program will request the next hiragana.

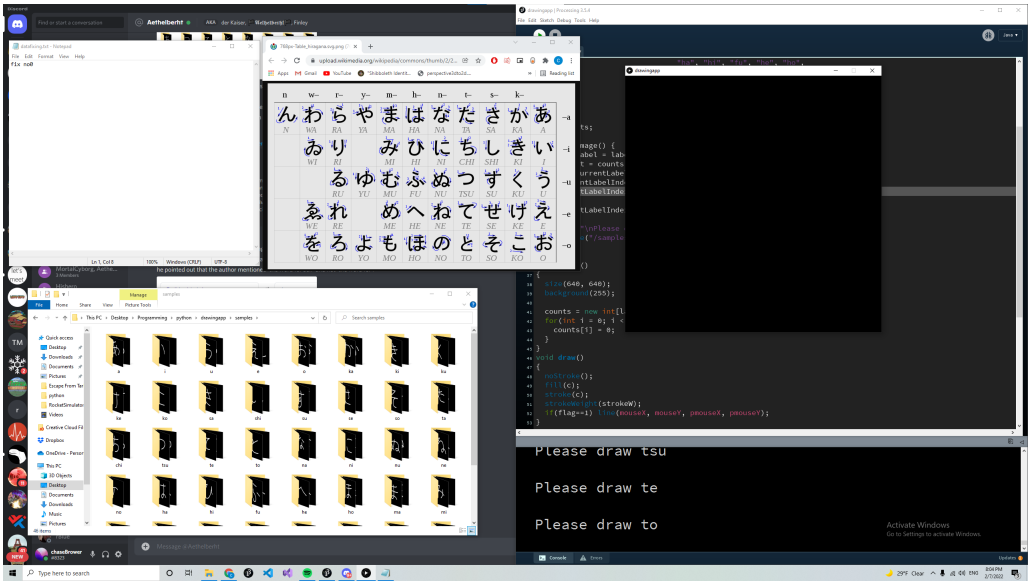


Figure 3: Process for data generation. Canvas to the right.

Using this method, characters are produced of consistent stroke width and size, on binary gray-scale. This means very little pre-processing needs to be done for the images to be usable.

2.2 Data Distribution

Through the aforementioned method, exactly 22 data were created for each class. Given 46 classes, there were 1012 images created in total. After inspecting the set, I found that 6 characters were incorrect or poorly written. I removed all of these images from the data-set. The resulting distribution can be seen in **Figure 4** below.

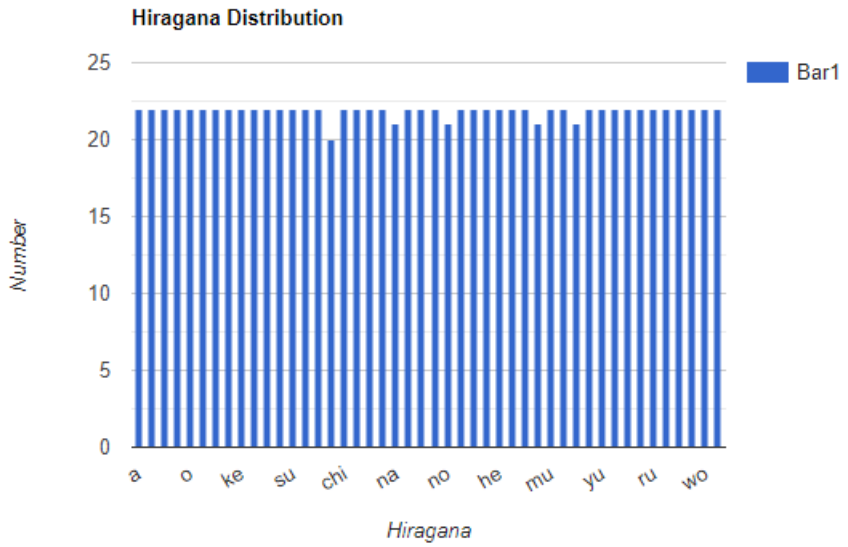


Figure 4: Distribution of data. Most labels omitted.

2.3 Data Normalization

In my notebook I tested most a number of image pre-preprocessing methods included in Keras. The characters are in high resolution compared to what the network needs (640 x 640), so I will downscale the characters to something around 160 x 160. Besides this, the data-set is already uniform and in good format, so I plan to run the downscaled images directly through ImageData-Generators with some small variation parameters.

3 Building an Overfit Model

The first step in building the predictive model was to build an overfit model to demonstrate that the architecture worked and was capable of learning the data. I began with a template model (see **Figure 5**).

```
Model: "sequential_2"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d_3 (Conv2D)           (None, 148, 148, 64)      1792

 max_pooling2d_2 (MaxPooling  (None, 37, 37, 64)       0
 2D)

 conv2d_4 (Conv2D)           (None, 35, 35, 32)        18464

 max_pooling2d_3 (MaxPooling  (None, 8, 8, 32)         0
 2D)

 conv2d_5 (Conv2D)           (None, 6, 6, 16)          4624

 flatten_1 (Flatten)         (None, 576)               0

 dense_2 (Dense)             (None, 10)                5770

 dense_3 (Dense)             (None, 10)                110

=================================================================
Total params: 30,760
Trainable params: 30,760
Non-trainable params: 0
_________________________________________________________________
```

Figure 5: The template model used for overfitting.

I carried out several experiments varying the parameters to make the model 'minimal'– to make it converge to a training accuracy of 1 with the least number of parameters. I finished with a model that had 5,554 parameters, and its learning curve (see **Figure 6**) still converged to 1.

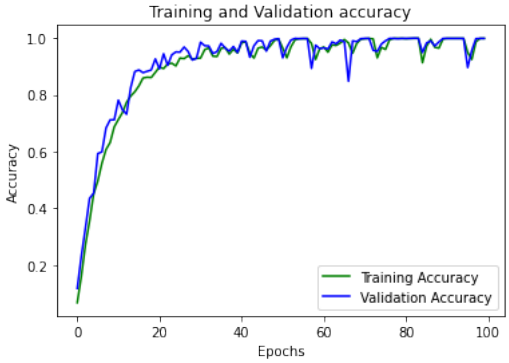


Figure 6: The learning curve for the final overfit model.

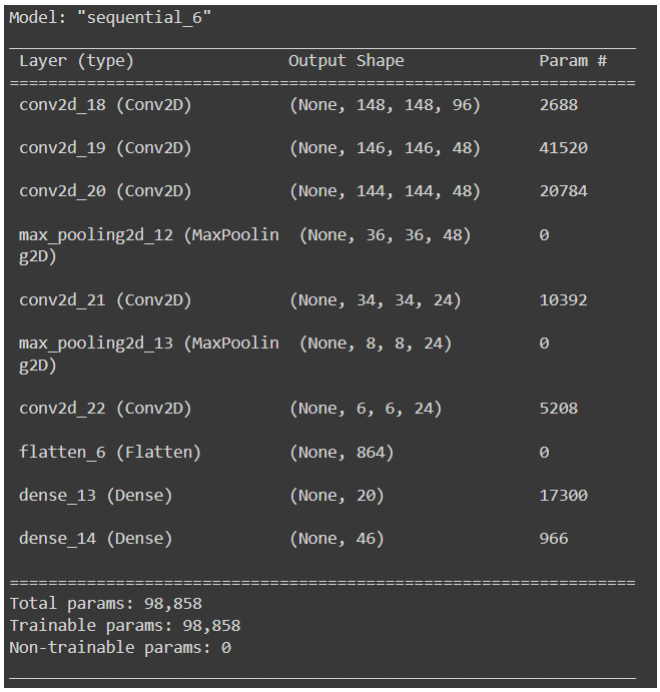
4 Splitting the Data

After creating an overfit model, the data is then split into training, validation, and test data. Training data are used for the model's regression, validation data are used to determine how effectively the model can generalize to other data, and test data are reserved to test the final model. The training/validation/test split can be used to verify that a model will continue to be accurate when it is moved to production.

For this project, the data were split into 80% training data, 10% validation data, and 10% test data. Deep learning models are very hungry for data for regression and training, so the bulk of the data is designated for training. After splitting the data, the model was varied under several experiments to maximize validation accuracy. Finally, the best-performing model (see **Figure 7**) was evaluated on the test data and its accuracy was measured. The best-performing model achieved accuracy, precision, f1, and recall scores of 93.25%, 95.63%, 93.38%, and 93.16% respectively. The training curve is provided in **Figure 8**.

5 Data Augmentation

To improve the previous model, data augmentation techniques were used. Data augmentation involves the automatic generation of new data based on permutations of collected data. For example, one would expect an image of a dog should still be recognized as an image of a dog even if the image is scaled up, rotated, flipped, cropped, or applied any number of other transformations. This allows much more data to be created to train the model, and is effective for preventing overfit issues. After running experiments including different rotation, shift, shear, zoom, and flip parameters, a model was selected using all parameters except for flip. The model achieved a validation accuracy of 98.9% at best. The training curve (**Figure 9**) is notable for its tightly coupled training and



```
Model: "sequential_6"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d_18 (Conv2D)          (None, 148, 148, 96)      2688

 conv2d_19 (Conv2D)          (None, 146, 146, 48)      41520

 conv2d_20 (Conv2D)          (None, 144, 144, 48)      20784

 max_pooling2d_12 (MaxPoolin  (None, 36, 36, 48)       0
 g2D)

 conv2d_21 (Conv2D)          (None, 34, 34, 24)        10392

 max_pooling2d_13 (MaxPoolin  (None, 8, 8, 24)         0
 g2D)

 conv2d_22 (Conv2D)          (None, 6, 6, 24)          5208

 flatten_6 (Flatten)         (None, 864)               0

 dense_13 (Dense)            (None, 20)                17300

 dense_14 (Dense)            (None, 46)                966

=================================================================
Total params: 98,858
Trainable params: 98,858
Non-trainable params: 0
_________________________________________________________________
```

Figure 7: The final model for split data.

validation accuracy curves–indicating lower overfit.

6 Data Regularization

Data regularization can improve a deep learning model in a few ways. Dropout involves removing parameters from the network during execution, which helps the model not to rely on highly specific features in the data to learn. Other methods including batch normalization and L1 and L2 regularization can address problems like vanishing gradient and ensure that the model is able to learn most effectively. In the case of my model, I was unable to improve the test performance with any types of regularization. The best model resulted in a test accuracy of 90%. An example of the model's training curve with a dropout rate of 0.1 is included (**Figure 10**).

7 Using Pre-trained Models

Pre-trained models in deep learning are usually massive models with numerous parameters, trained at an enterprise data center. Such models include aggregations of many data types and labels that can then be adapted to a diverse range

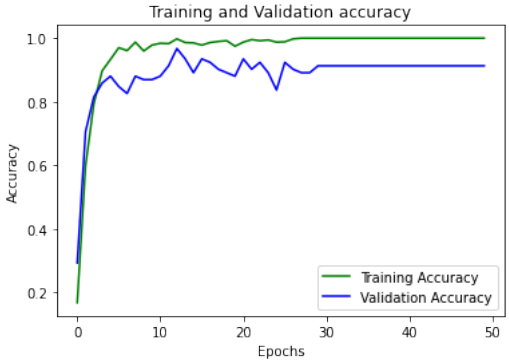


Figure 8: The training curve for the split data model.

of problems. Despite not having been hand-tailored to a problem in question, the model is effectively able to represent many types of data with its convolutional layers, and a small group of dense layers can often learn to use that representation for a particular problem. For my model, I used the VGG16 and ResNet50 models with a few sizes of dense layers following. Unfortunately, the data involved in my problem are computer-generated and not likely to be represented in mass data sets. After 4 hours of training, the ResNet50 model had achieved a test accuracy of 70% and had slowed down significantly (training curve pictured in **Figure 11**). The VGG16 model was unable to learn.

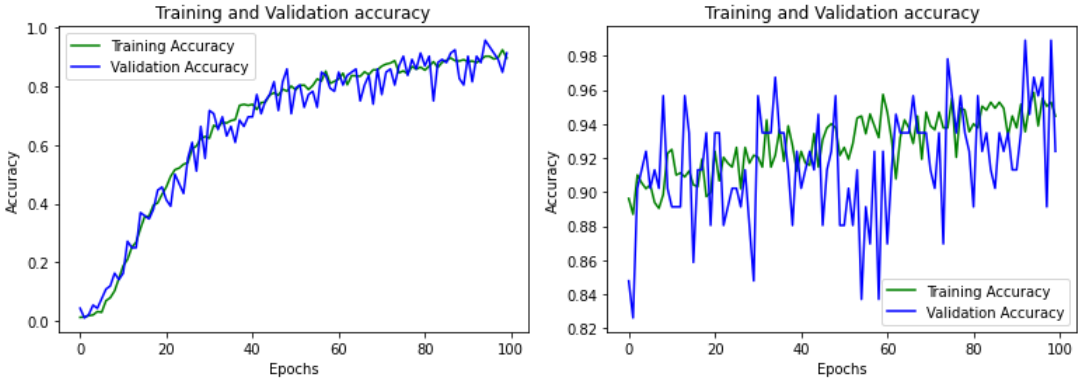


Figure 9: Training curve for augmented model. First 100 epochs to the left and the next to the right.

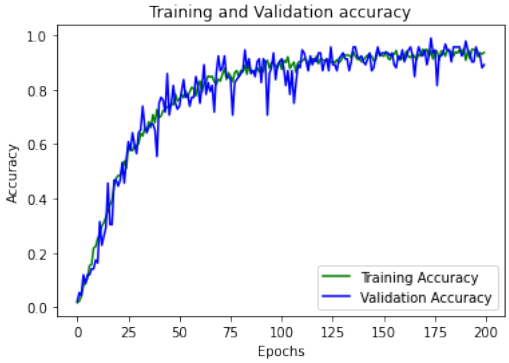


Figure 10: Training curve for the dropout model.

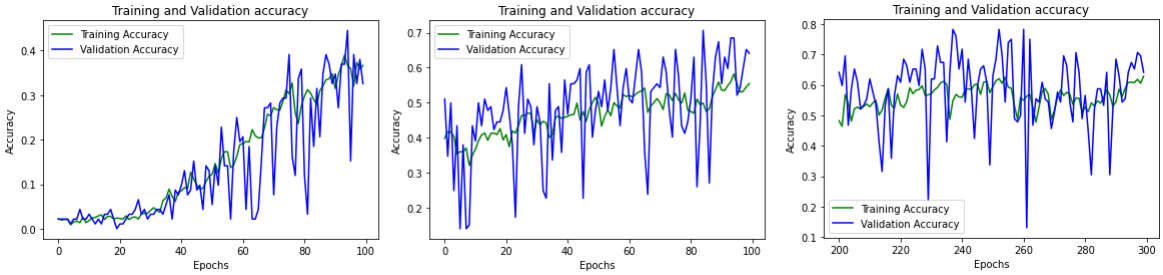


Figure 11: The training curve for the ResNet50 architecture.

8 Conclusion

After six phases of model development–data gathering, overfit, generalization, augmentation, regularization, and pre-trained models, the model was able to effectively recognize the Japanese Hiragana. Gathering computer-generated data meant that the model was easy to train initially, but it resulted in some drawbacks like in the in-applicability of pre-trained models. The best model across all architectures was an augmented model with a validation accuracy of 98.9%. Augmentation proved particularly useful in improving the accuracy of the model by producing a large dataset that could effectively avoid overfitting issues. In further iteration, it may be worth exploring regularization to improve the model.

A prototype production model was built in processing (based on java) to test the model's performance in real time. The program (**Figure 12**) allowed the user to draw a character, and a python script would periodically scan the drawn image to detect the character. This format allows for better diagnosis of the behavior of the model. For example, a peer was able to identify that the model became ineffective if a character was drawn in the corner of the frame. In future iterations, it may be beneficial to apply more aggressive shift and scaling parameters during augmentation, or collect more varied data, to ensure that the model is effective in all cases. This process revealed some facts about the model that accuracy and other metrics are too vague to explain. This further highlights an important lesson in this project and in deep learning in general–the focus should always be on understanding the model rather than 'shallow optimization.'

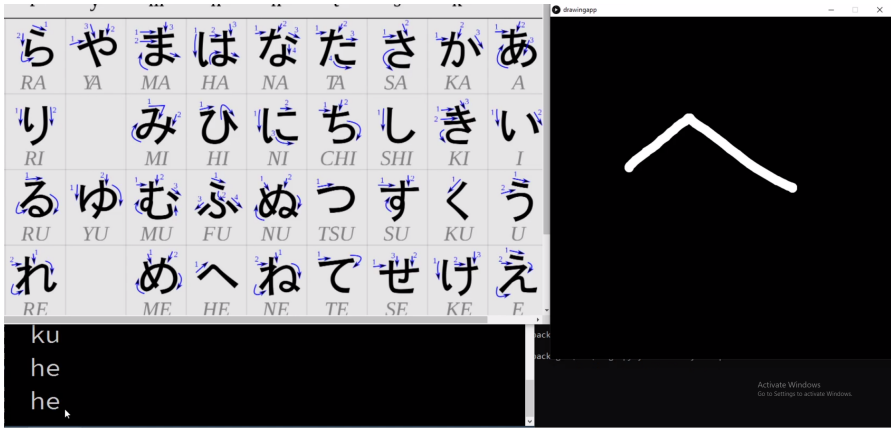


Figure 12: Image of the prototype program used for real-time model prediction.