A Neural Network to Recognize Number of Fingers

John Berroa

April 14, 2018

Contents

Abstract

A neural network is created to recognize how many fingers are held up in images of hands. The dataset is proprietary and collected for the purposes of this experiment. Two model types are trained: a replication of a prior finger counting network, and a deeper multilayer convolutional neural network. Results show the CNNs reaching an accuracy of around 60% on the new data.

1 Introduction

In a prior class, *Comparative Machine Learning*, we used open-source code[1] called "How Many Fingers" (HMF) to recognize how many fingers a person held up in front of a camera. This code used a convolutional neural network (CNN), and preprocessed the hand images heavily by thresholding to extract the edges of the hands. Additionally, all images were taken against a pure white background with no noise. The network learned very well, and was even able to generalize to "non-standard" numbers, such as holding up the index and little finger to denote "two." The purpose of the current project is to to extend this network, making it more advanced and also robust against different backgrounds. Therefore, it was decided to create a deeper convolutional neural network architecture, and capture a proprietary dataset of hand images to train on[1]. This would require a data collection script, a data loading script, and the models[2]. The following paper is laid out as follows: first, the dataset collected will be described in Section 2, second, the models used to train on the images will be discussed in Section 3, third, the results of the networks' performance will be reported in Section 4, and finally some remarks to conclude will be mentioned in Section 5.

2 Dataset

In total, 2110 images were collected, with equal amounts of image per category (one though five fingers).

2.1 Image Contents

Images captured were of size 300x300 pixels. A few other people provided some pictures of their hands, but most of the data is predominantly the author's hands. Pictures have varying backgrounds, but backgrounds remain similar during each "collection period," i.e. a run of taking images. Also, the white walls of the author's apartment have a disproportionate showing in the data (this can be seen in Figure 2.1). Since there is a difference between the American way to show numbers and the German way, both styles for numbers two, three, and four

[1] Dataset available upon request.

[2] All source code and this report is available on GitHub: `https://github.com/johnberroa/Finger-Counting-Neural-Network`

were captured (for some reason I forgot "one" is just the thumb in Germany). If the model performs like the HMF model, it really shouldn't matter because it will have the ability to generalize to the German way; however, this way builds that robustness directly into the model. Images were resized sometimes to deal with memory consumption. Comparisons between the two datasets are shown in Table 2.1.

Comparison between HMF and Our Data

Metric	HMF	Ours
Amount	approx. 2100	2110
Size	300x300	50x50/300x300
Channels	1 (b&w)	3 (color)
Content	Thresholded binary image with blank backgrounds	Color image with noisy backgrounds
Labels	0–5	1–5

Table 1: Data characteristics

2.2 Collection

Images were captured from a 300x300 pixel patch via the author's laptop's webcam. Upon starting the script, the user is greeted with short instructions, and is prompted to input the total number of images desired to capture, as well as the starting label. Pressing the spacebar captures an image, and pressing the number keys changes the label of the image captured. Images are saved as .jpgs with the naming convention *label-number.jpg*. Average image size on disk is 30 KB. The screen displays useful information regarding the data acquisition process: number of images captured, current label, and an indicator when to switch labels (if equal amounts of images per label is desired). Upon conclusion of data capture, the script moves all images into their respective folders. The file structure is shown in Figure 2.2. This data structure was chosen for a few reasons:

1. It allows easy exploration of specific labels to check quality of data.

2. Allows easy renaming if data is deleted.

A further explanation on point two: if some data collected is not good, e.g. wrong label, then deleting them will ruin the numbering of the data. Renumbering would be a nightmare if all images were in one folder, so with them being separate, it makes things easier. Numbering is a problem because when loading the data, the script steps through each number appended to the filename, and if an image that is supposed to have a certain number isn't there, it could cause problems. Thinking back, this could have been resolved with a try/except, but the current solution works just as well. Data is occasionally backed up (or can

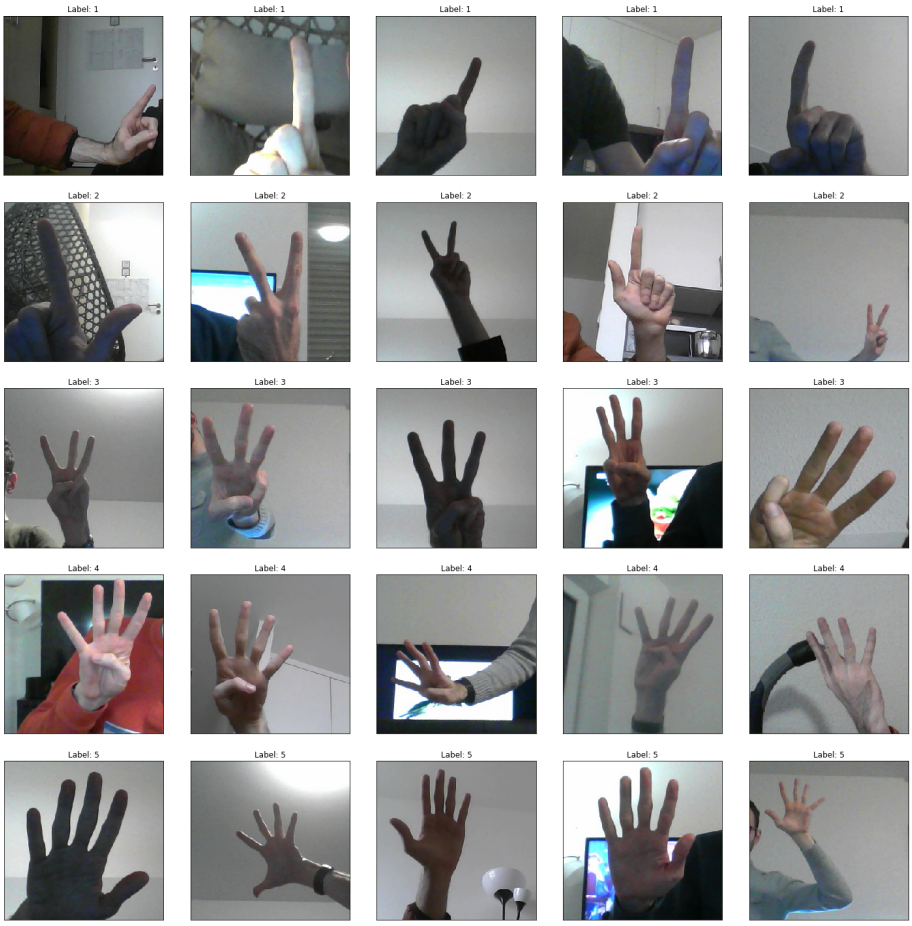

Figure 1: Example images from the dataset

```
Finger Neural Network
├── data_collection.py
├── data_loaders.py
├── backup.py
├── 1
├── 2
├── 3
├── 4
├── 5
├── tmp
├── FINGER DATA BACKUP
└── ...(Keras, summaries, etc.)
```

Figure 2: Directory tree

be forced manually to) with the `backup.py` script. This copies all folders into the backup folder.

The script can be seen in `data_collection.py`.

2.3 Data Loading

There are two data loading methods. The first is loading all the images and splitting into 60% train, 20% validation, and 20% test. The other is to use Keras' inbuilt image generator functions. This split that data into 200 images to train per class, and 100 images per class for validation and test.

2.4 Normalization

All Keras scripts normalize the images into the range $[0, 1]$.

2.5 Augmentation

Since a little over 2000 images is not a large dataset by CNN standards, the data is augmented heavily. Keras image generators were used to flip, crop, rotate, and shift images on the fly. Random flips over the vertical axis are included because that would essentially replicate taking pictures of both the left and right hands. This would then eliminate any handedness bias in the data (if through random permutations both "classes" end up equal in size). In effect, through the random flips, the dataset is doubled in size. With the other augmentation techniques, it is hard to say, but conservatively one can say at least another doubling. So the effective size of the fingers dataset is above 8440 images.

3 Models

First to test was a simpler model, a multilayer CNN to see if the task could even be learned with the more information rich data. This was done in Keras.

For model comparison, three other models were tested. First, a support vector machine (SVM), created with sklearn. Then two CNNs: the first was a replication of the CNN used in HMF, and the second was a bigger design to see how much performance can be achieved from a conventional CNN. Both CNNs were made in Keras. It is important to note that, with five classes, chance level is 20%.

3.1 SVM

A support vector machine model was fitted to show whether or not neural networks have any advantage in this task. The sklearn implementation was used, and various values for the kernel coefficient (gamma) were tested.

The code for this network is in `keras/finger_svm.py` (because it makes use of the *tmp* folder that Keras uses).

3.2 Neural Network Architectures

3.2.1 Simple CNN

The simple CNN's purpose was to see if the task was learnable by a neural network.

It had the following network structure: Two ReLU activated convolutional layers with 4 and 8 filters of size 3x3, respectively. Each are followed by a 2x2 strided max pooling. The images are then flattened and sent into a dense feedforward layer with 100 neurons and ReLU activation. Dropout is then performed with a keep probability of 50%. Lastly, the data is fed into an output layer with 5 neurons and softmax activation. The optimizer used is adam with categorical cross entropy loss.

After some hyperparameter tuning, minibatches of size 500 were used, and the network was trained for 500 epochs. There was no data augmentation.

The code for this network is in `keras/keras_finger_CNN.py`.

3.2.2 HMF CNN

This model was a replication of the HMF code, in order to see if it would achieve similar performance with a similar model, given more complex data. There is one difference: the output layer has 5 neurons instead of 6 (recall that the author included 0 as a label), and there was no thresholding preprocessing completed.

The network structure is thus: Four ReLU activated convolutional layers with 32, 64, 128, and 128 filters of size 3x3, respectively. Each are followed by a 2x2 strided max pooling. The images are then flattened and sent into a dense feedforward layer with 512 neurons and ReLU activation. Dropout is then performed with a keep probability of 30%. Lastly, the data is fed into an output layer with 6 neurons and softmax activation. The optimizer used is adadelta with categorical cross entropy loss.

A minibatch size of 128 images was used, and the network was trained for 500 epochs (but only the first 40 were comparable as per HMF's epoch count).

The code for this network is in `keras/keras_finger_HMF.py`.

3.2.3 Large CNN

To see how well a "normal" CNN can perform on the task, some extra layers were added and normalization plus layer L2 regularization (since there were major overfitting issues without it).

The "large" network had the following structure: Five ReLU activated convolutional layers with 16, 32, 64, and 16 filters of size 3x3, respectively. The outputs of the convolutions are batch normalized, and then followed by a 2x2 strided max pooling. The images are then flattened and sent into two dense feedforward layer with 512 and 128 neurons and ReLU activations. Dropout is performed between the feedforward layers with a keep probability of 50%. Lastly, the data is fed into an output layer with 5 neurons and softmax activation. The optimizer used is adam with categorical cross entropy loss.

50x50		Full Sized	
Gamma	Accuracy	Gamma	Accuracy
default	36.25%	–	–
.001	42.29%	.001	41.58%
.01	54.50%	.01	20.73%

Table 2: Performance of the SVM

After some hyperparameter tuning, minibatches of size 256 were used, and the network was trained for 500 epochs.

The code for this network is in `keras/keras_finger_LargeCNN.py`.

4 Results

4.1 SVM

The SVM was fitted with full sized and 50x50 rescaled images. The hyperparameter tuned was *gamma* (kernel coefficient), with which various values were tried. The validation accuracy results are summarized in Table 4.1. It is interesting to see that the gamma values that work well on the smaller images don't work well on the full sized ones.

Interestingly, when using the full size images, the best gamma of .01 on the smaller images leads to a validation accuracy of 20.73%...almost perfectly chance.

4.2 Neural Network Architectures

Following are the performance results of the neural network architectures. In summary, the simple CNN extremely overfit, the HMF CNN and large CNNs overfit but achieved roughly the same accuracy. The notebook to generate the graphs can be found in `keras/Graphing Notebook.ipynb`.

4.2.1 Simple CNN

Using smaller 50x50 images, this model reached an accuracy of 98.73% on the training data after 1000 epochs. However, the validation accuracy only reached 59.24%. Looking at the loss plots (see Figure 4.2.1), it is obvious why this happened: there is severe overfitting after the first 18 or so training steps. Regardless, the purpose of this network was to see if the data can be learned, and since the almost 60% accuracy far exceeds the 20% chance level, this network was a success.

However, using the full 300x300 images yielded chance results with smaller sized (i.e. 32, 128) minibatches. With a minibatch of 500, the network did learn on the full sized images. However, the loss and accuracy curves exhibit a very interesting shape: it seems that with a 500 minibatch, chance level is still

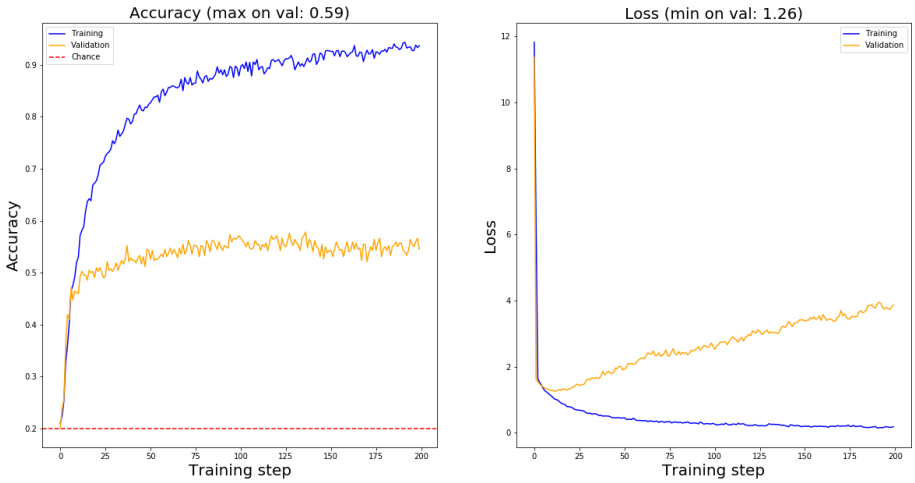

Figure 3: Performance of the simple CNN on 50x50 images

possible, but this particular run was lucky to end up finding a minimum and fall into it. It ended up with a 51% validation accuracy, and the curves can be seen in Figure 4.2.1.

4.2.2 HMF CNN

Because the main goal is to try and replicate (and extend) the HMF model, see Figure 4.2.2 showing 40 epoch comparisons of both networks.

The model reached 100% on the training data, which clearly indicates overfitting. This is confirmed with the validation accuracy, which peaks at 57.22%. This overfitting also occurs very early in the training process, like the simple CNN on the small images.

4.2.3 Large CNN

The "large" CNN achieved pretty much similar results as the HMF CNN. This might be because adding more parameters to a problem that can be described with fewer parameters doesn't help much. Accuracy also hovered in the 60% area, and the graphs looked essentially the same as the HMF graphs (omitted for space).

5 Conclusion

One of the major conclusions (and lessons learned) from this project is how challenging it is to train a CNN on non-toy problem datasets. In the beginning, there were quite a few "out of memory" errors. With that solved, trying to solve the overfitting issue was next. The small dataset as well as the more complex

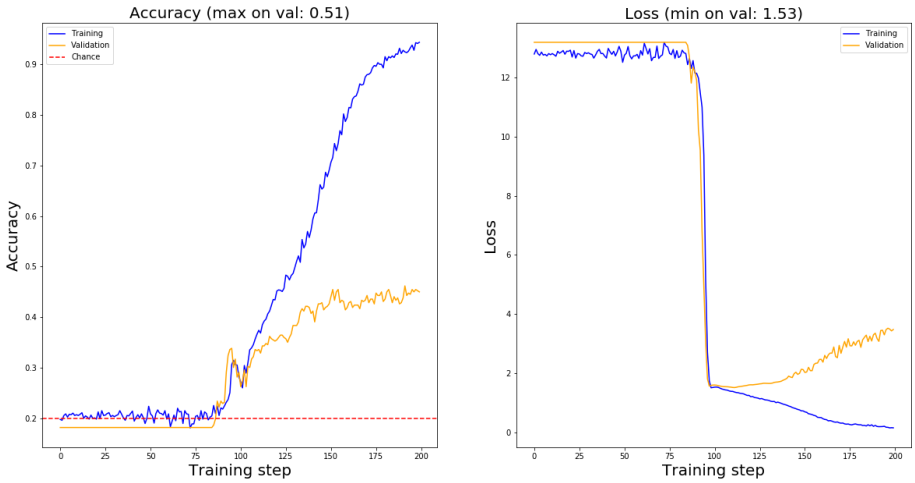

Figure 4: Performance of the simple CNN on full size images (500 minibatch size)

images definitely had an effect on the capability of the network, considering that the original HMF network did achieve almost perfect accuracy on its thresholded images.

This raises another important lesson: network architecture tuning isn't as important as feature extraction. Further work with this network would be to apply the same feature extraction as the HMF author, in order to "help" the network along. It was originally expected that the network should figure out how to extract these features on its own, but trusting that to happen is not a good idea.

Other convolutional architectures would also be interesting to try, such as a ResNet architecture. It was attempted to apply ResNet [2] to this problem, but as of the deadline, issues with dimensionality explosions were not resolved.

References

[1] jgv7, "Cnn-howmanyfingers." GitHub repository `https://github.com/jgv7/CNN-HowManyFingers`, 2017.

[2] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778, 2016.

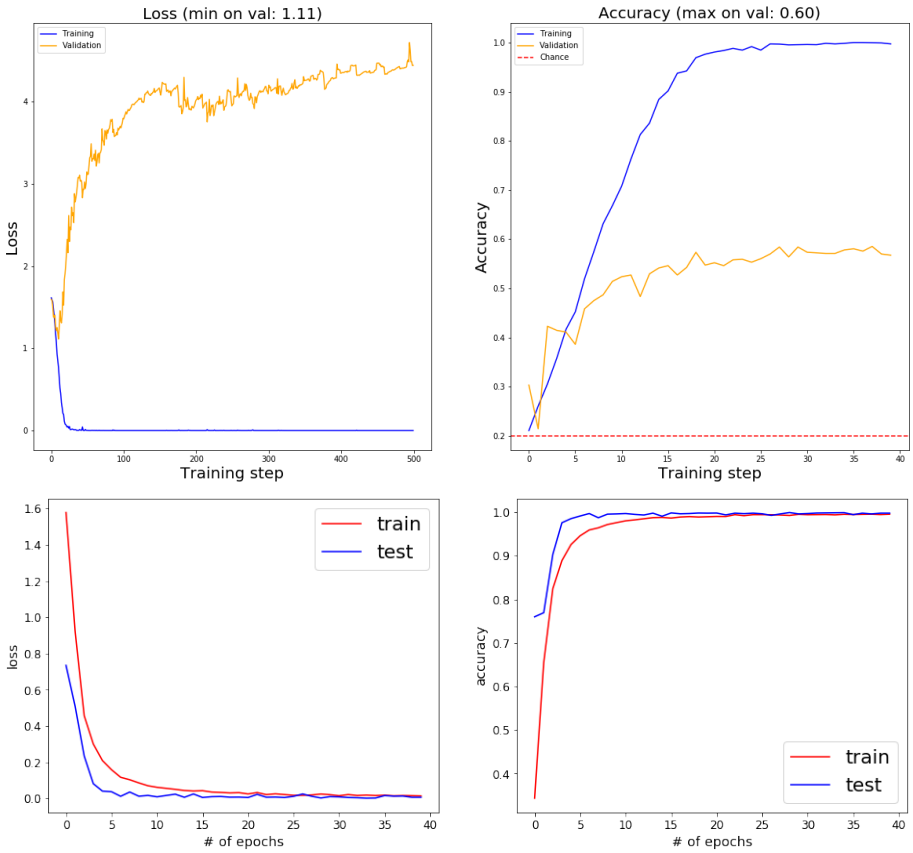

Figure 5: Comparison between both original and our HMF networks at 40 epochs