Scene Text Recognition With CNNs

Alexander Moulton, Ashley Famularo, Daniel Mello

May 07, 2021

1 Abstract

The use of convolutional neural networks has become commonplace in the realm of computer image recognition. There have been several datasets that have been difficult to make reliable networks to recognize, however, given the presence of background noise and images taken at various angles. One such dataset is the Street View House Numbers (SVHN), which has many of these issues. This is highly prevalent in many examples of scene text detection, both in detecting the presence of text and then classifying what it says character by character. In this study, we focus on recognizing single numbers extracted from images of houses taken from Google Street View. These images have pieces of other numbers in them as well as various skewing due to the angle the picture was taken and various background colors. We implement a CNN and experiment with various parameters and helper functions to examine their effect on mitigate these noisy background features.

2 Introduction

A convolutional neural network, or CNN, is defined as a deep learning neural network designed for the processing of structured arrays of data. These types of networks are widely used in computer vision and are primarily reserved for many visual based applications such as image classification. They have also found success in natural language processing for text classification [2]. CNNs have a reputation for being good at recognizing patterns in the data that they are provided, specifically lines, gradients, circles, and even eyes/faces. It is this that makes them a powerful tool in the realm of computer vision. They differ from earlier computer vision algorithms as they work directly with raw image files, removing the need for the files to be preprocessed before the network could work with them. Convolutional networks are feed-forward networks, and are often made up of twenty to thirty layers plus they contain the special convolutional layer. Convolutional layers stack on top of one another, each one capable of recognizing more complex shapes. With just three or four of these layers it is possible to recognize handwritten digits and with twenty-five of them it is possible to distinguish human faces [2].

Scene text is a term that is used to describe text found in natural locations. Natural locations are limited to outdoor environments, so on places like street signs, billboards, banners, etc. are considered natural locations. These are chosen over computer generated text as they

feature varying lighting and focus making it more difficult to identify each symbol which makes for better training on the model instead of constant and uniform lighting/focus.

3 Related Work

There have been previous CNN models constructed for the purpose of detecting and converting text from images into plaintext readable by software. Such examples include Tesseract OCR and gOCR, two OCR (Optical Character Recognition) technologies designed to read and convert scanned documents into editable text or PDF files. However, such technologies typically implement models trained on datasets containing images of clearly readable text against a flat background color, thus they do not accurately depict identical text located in natural environments. Because of this, these technologies cannot easily be adapted to the recognition of scene text without the application of additional transformations via some image manipulation library such as OpenCV.

Advancements in the field of scene text recognition have been made in the way of detecting instances of characters as opposed to the actual reading of characters from images. Such examples of these technologies include EAST, a text detection algorithm implemented by the OpenCV library, and CRAFT, a similar detection algorithm made available by PyTorch. The CNN we have developed and trained via the SVHN dataset would be implemented in such a way as to process text detected in images via these algorithms and make them readable to software systems for various purposes.

4 Problem Definition

The purpose of this project is:

- Create a model capable of recognizing scene text in natural environments.
- Accurately identify examples of scene text in images that present obstructions and varying modification made to text.
    - Modifiers include: focus levels, zoom, rotations, light levels, height differences, textures.

5 Dataset

For training our model we first thought to use the Modified National Institute of Standards and Technology or MNIST database as it contained sixty thousand training images which made it have lots of variation and would be a challenge for the neural network to learn from. However, this data contained mostly only handwriting which can be very different from actual scene text that would be put against the trained model. So that dataset was scrapped and we moved on to The Street View House Numbers or SVHN dataset. The SVHN is a real-world image dataset for developing machine learning and object recognition algorithms with minal requirement for data preprocessing and formatting. This dataset can be seen as similar in flavor to the MNIST dataset, but it incorporates an order of magnitude more labeled data and comes

from a significantly harder, unsolved, real world problem [1]. While the dataset advertises that it has over six hundred thousand images, this was a bit more than what we need for our model. The dataset was broken down into three different matlab files, one for training which contained fifty-eight thousand six hundred and five image, one for testing which contained fourteen thousand six hundred and fifty-two images, and lastly the additional or extra set which contained five hundred and thirty-one thousand one hundred and thirty-one images. However, the extra set was not utilized for this experiment.

6 Methods and Methodology

Python was chosen for this project for its well-developed APIs from machine learning. Before the model could be created and trained, it was necessary to import and process the data. Given that the datasets were in a matlab (.mat) file, SciPy's `io.loadmat` function was used to load the data into a dictionary with variable names as keys and matrices as values. It was then possible to make these dictionaries into NumPy arrays by splitting them based on X and y for test and training. The "X" values (the images) of both the train and test datasets were then cast as 64-bit float values. The "y" values (the labels) had to be flattened as they were originally in a 2-dimensional array. **Figure 1** shows the dataset shapes before and after this transformation. Additionally, the pixel values were normalized to between 0 and 1. Also witnessed in the image, the pixel values come before the image index in the X arrays, meaning in order to return the 32x32x32 image, one would need to refer to the array as `X[:,:,:,i]` to return the image at the ith index [1]. It is much simpler to simply rearrange the array, so the ith index could be referred to the same way as the y values. This process was tested by printing an image and its corresponding label. **Figure 2** both demonstrates the increased usability of the rearranged array reshape and verifies its correctness.

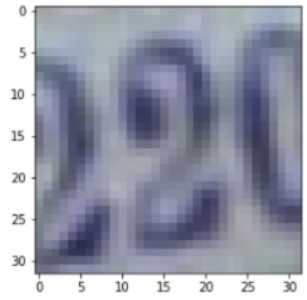
Fig. 1 - Dataset Shapes Pre/Post-Transformation

```
Before:
X Train Shape:  (32, 32, 3, 73257) | y Train Shape:  (73257, 1)
X Test Shape:  (32, 32, 3, 26032) | y Test Shape:  (26032, 1)
Pixel data type:  <class 'numpy.uint8'>

After:
X Train Shape:  (32, 32, 3, 73257) | y Train Shape:  (73257,)
X Test Shape:  (32, 32, 3, 26032) | y Test Shape:  (26032,)
Pixel data type:  <class 'numpy.float64'>
```

Fig. 2 - Transformed Data Verification

```
plt.imshow(train_imgs[2222])
plt.show()
print('Label: ', train_lbls[2222])
```




```
Label:  2
```

Although not completely necessary since the classifications are all numeric, the label sets were one-hot encoded using Scikit-Learn's `preprocessing.LabelBinarizer`. This would be completely necessary if these images and labels had any other characters such as letters or symbols, so it has been included as a precursor to such a model. The last step before the model creation is to split the training images and labels into training and validation sets. This was accomplished with Scikit-Learn's `model_selection.train_test_split`. It's fairly standard to use an 80%-20% split for

the training and validation sets respectively. A random state was also used to mix the data and was initially set to 0 arbitrarily. This number can be altered to influence the order in which data is fed to the model, sometimes leading to different results.

Keras was chosen as the machine learning API to build the model as it provides a simple interface for stacking layers, defining hyperparameters, integrating callback and utility functions in training, creating optimizers, and extracting training data. Figure 3 shows the general initial model layer layout first tested, however, before this, a model configuration of **Convolutional→BatchNormalization→Convolutional→MaxPooling→Dropout** was repeated three times before **Convolutional→MaxPooling→Flatten→Dense→Dropout→Softmax**. This model never reached over 11% training accuracy, however, and was later altered.

Two callback functions were then defined: Keras's `callbacks.EarlyStopping`, and `callbacks.ReduceLROnPlateau`. The former was implemented to prevent overtraining of the model based on the lack of improvement of a defined metric. This allows epochs to be set high with no consequence. The second of these functions checks for the lack of improvement in a metric and, if it hasn't improved for a set number of epochs, lowers the learning rate by a scalar. Both of these metrics were set to monitor training loss. Early stopping was originally set to end training after 8 epochs without at least a 0.001 drop in training loss. Learning rate was, perhaps aggressively, set to be reduced by half if loss wasn't improved from 1 epoch to the next until it reached a floor of 1e-7.

Further variation in the training data was achieved with the implementation of Keras's `preprocessing.image.ImageDataGenerator`. This was left out of the first implementation to see if it was even needed, but it augmented the dataset by performing rotation, height, shear, and zoom transformations. This was implemented but not utilized so early in the process because the dataset already has many images that already have different heights, rotations, shears, and zooms. If the dataset was made more robust in these variations, the model will likely be able to recognize images that have more outlying features in these categories.

The model was then compiled using a standard Adam optimizer while tracking categorical cross entropy loss. The standard metric to be evaluated to be evaluated by the model was set to "accuracy". A summary of the evolution of the model structures and their corresponding training results can be found below.

The initial iteration of our CNN architecture begins with a set of two convolutional layers, each followed by a max pooling layer. The first convolutional layer has an output filter size of 32 and a kernel size of 3 by 3. The output of this layer feeds into a max pooling layer with a kernel size of 2 by 2. The second convolutional-maxpool pair is identical, with the only change being an increase of the convolutional filter size from 32 to 64. The second max pooling layer feeds into two sequential convolutional layers, each having an output filter size of 128 and a kernel size of 3 by 3. The output of these two convolutional layers feeds into a max pooling layer with a kernel size of 2 by 2, which feeds into a dropout layer with a dropout rate of 0.3 to prevent overfitting. Once the initial dropout is performed, a flatten layer reduces the dimensionality of the data to make it readable by a dense layer with a filter size of 128. A second dropout layer

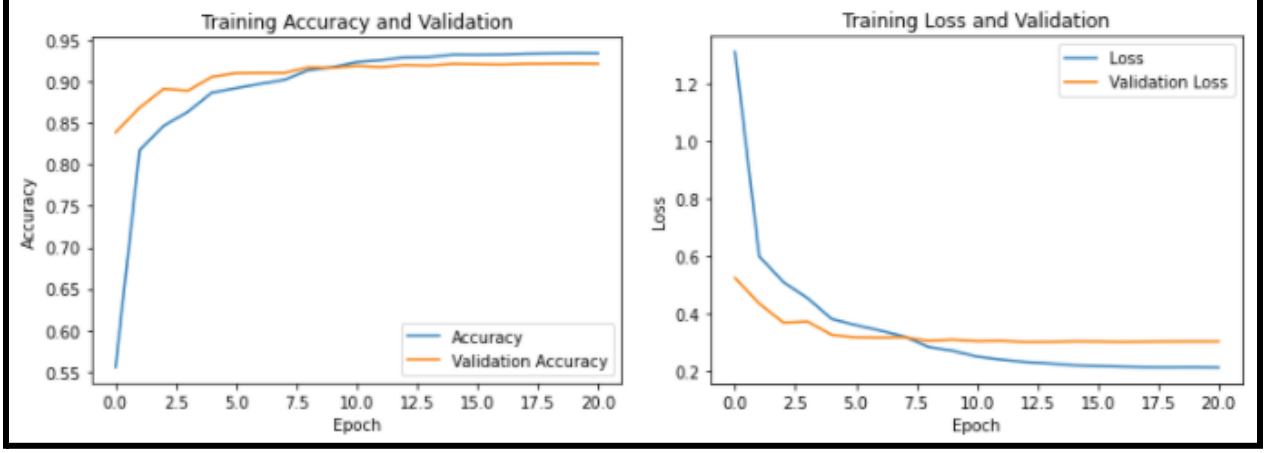
with a dropout rate of 0.4 is applied to further reduce overfitting. Finally, a softmax dense layer with a filter size of 10 is applied, with each output filter being representative of the 10 digits attempting to be classified (0-9). It should be mentioned that, as previously stated, the ImageDataGenerator was left out of this iteration and a standard adam optimizer was used. This model used a batch size of 50 over 30 epochs.

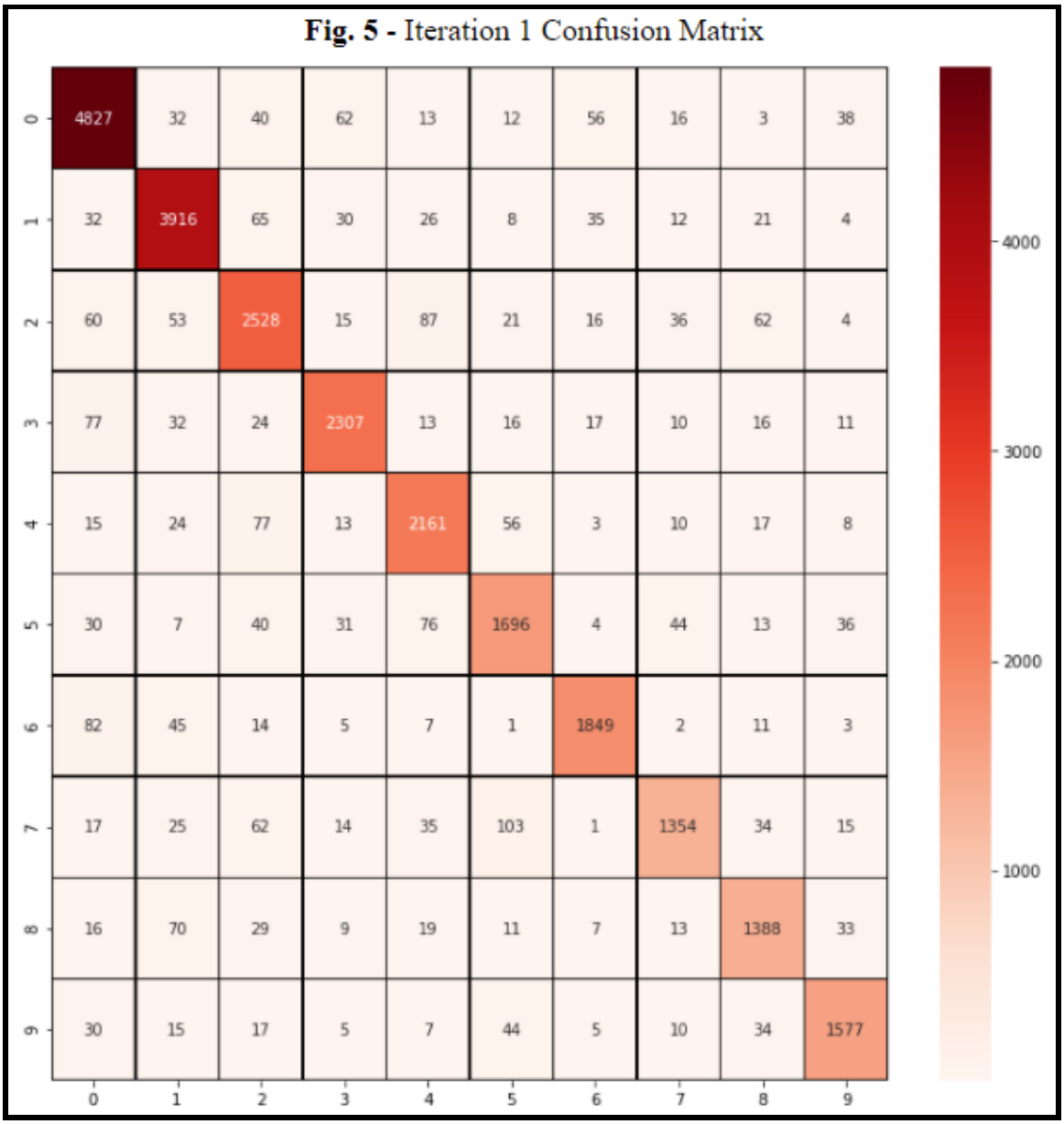
Fig. 3 - Iteration 1 CNN Architecture

Layer Type	# of Filters / Dropout Rate	Kernel Size	Activation Function
Convolutional	32	3 * 3	ReLU
Max Pool	-	2 * 2	-
Convolutional	64	3 * 3	ReLU
Max Pool	-	2 * 2	-
Convolutional	128	3 * 3	ReLU
Convolutional	128	3 * 3	ReLU
Max Pool	-	2 * 2	-
Dropout	0.3	-	-
Flatten	-	-	-
Dense	128	-	ReLU
Dropout	0.4	-	-
Dense	10 (Output)	-	Softmax

As seen in **Figure 4**, by the time the first model iteration had completed training, it had achieved a training accuracy of 93.4%. After validating the model, it was able to achieve a final accuracy of 90.6%. These results provided the benchmark by which we would compare future model iterations to. It's important to note that only 22 epochs were required to achieve an accuracy of 90.6%. For the next model iteration, it was assumed that increasing the number of epochs would increase the accuracy of the model. After training, the model's cross entropy loss stood at 0.21. After validating the model, the cross entropy loss stood at 0.34. Similar to the Training Accuracy/Validation, these results provided us with a benchmark to compare future results to. The functioning of the learning rate reduction can be seen following the oscillation between epochs 2-4.

Fig. 4 - Iteration 1 Training and Validation Accuracy and Loss





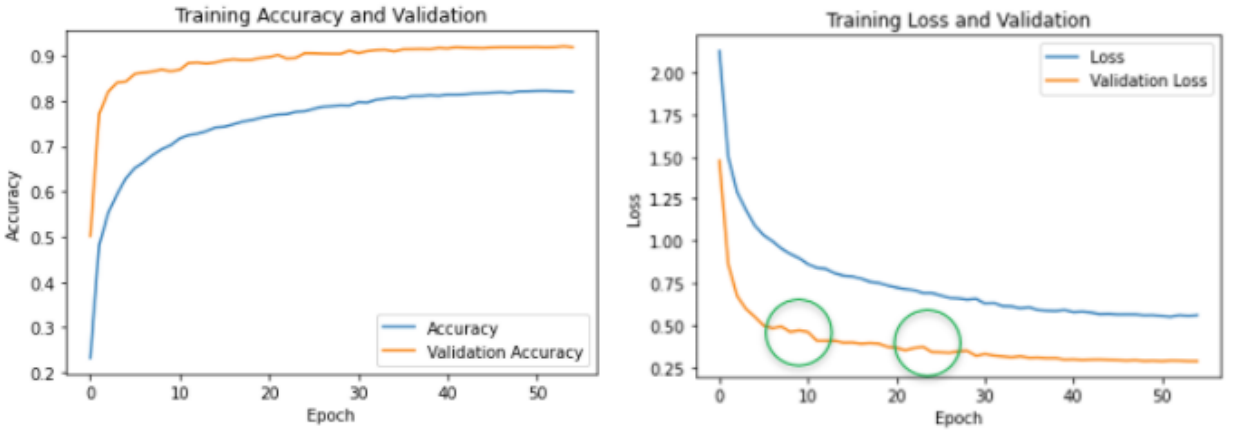
The confusion matrix for iteration 1 reflects the model's accuracy. The most commonly identified digit was "0". As the features of each digit grew in complexity, there was a dropoff in the number of correct guesses.

The second iteration of our CNN architecture begins with a set of 3 Convolutional-Maxpool-Dropout pairs. The kernel size for each convolutional layer is 3 by 3, and the filter size for each convolutional layer increases by a factor of 2 with a starting size of 32 and an ending size of 128. The dropout rate for each subsequent dropout layer increases by .1, starting with a dropout rate of .1 and ending with a dropout rate of 0.3. The output of the final dropout layer feeds into a flatten layer to reduce the dimensionality of the inputs. The flatten layer then feeds into a dense layer with a filter size of 128. The dense layer then feeds into a dropout layer with a dropout rate of 0.4, which feeds into another dense softmax layer with a final output filter size of 10. This model also utilized the ImageDataGenerator with 10 degree rotational variation, 20% of total height variation, 20% shear intensity variation, and 90%-110% zoom variation. An Adam optimizer was used, but created outside the compile method with an initial learning rate of 0.001 to include the experimental AMSGrad variant [3]. A batch size of 50 was used over 100 epochs. The early stopping function patience was reduced to 3 epochs to reflect the dramatic increase in epochs. The learning rate reduction was set to 75% of the original learning rate to more gradually decrease the change.

Fig 6. - Iteration 2 CNN Architecture

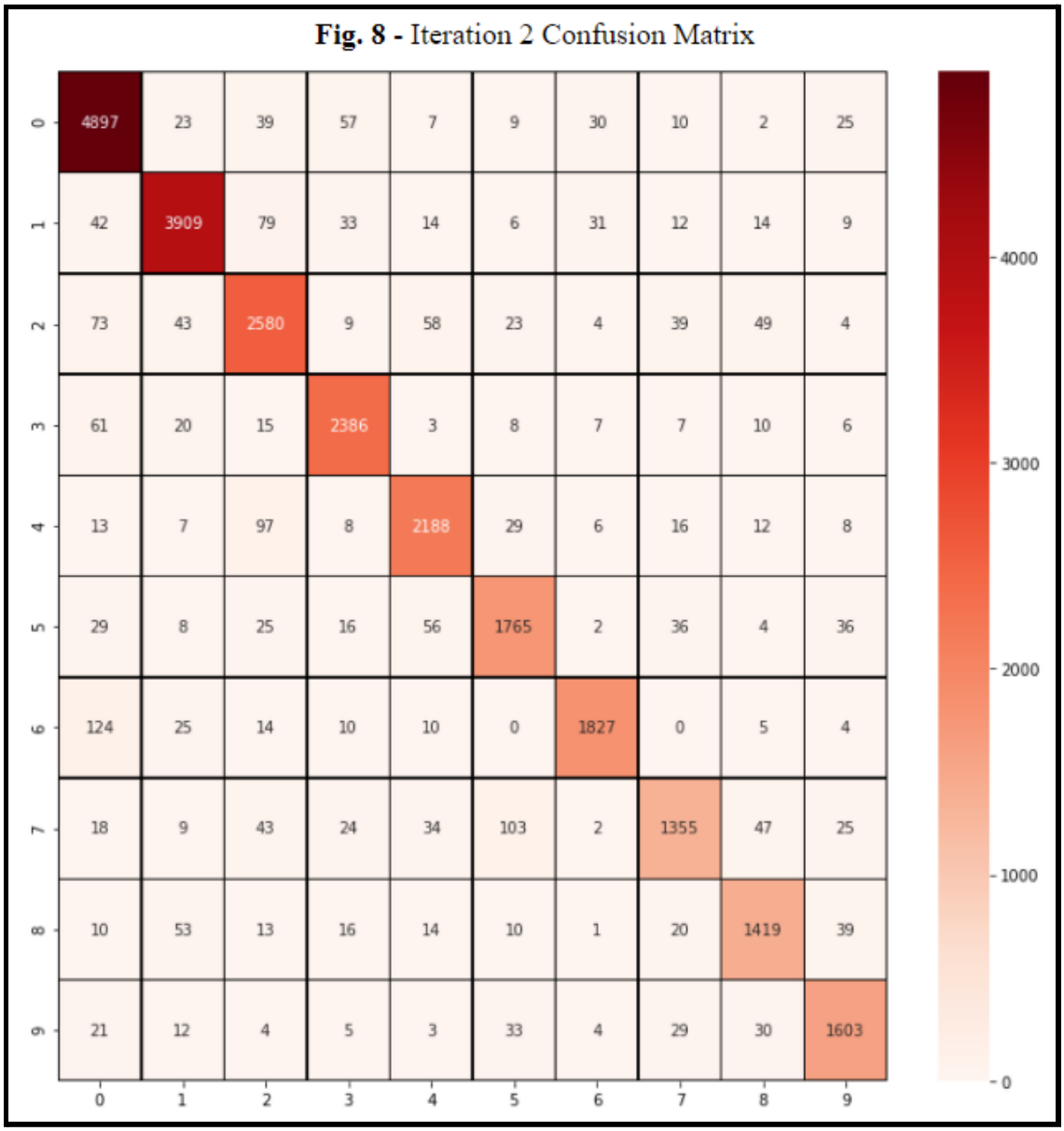
Layer Type	# of Filters / Dropout Rate	Kernel Size	Activation Function
Convolutional	32	3 * 3	ReLU
Max Pool	-	2 * 2	-
Dropout	0.1	-	-
Convolutional	64	3 * 3	ReLU
Max Pool	-	2 * 2	-
Dropout	0.2	-	-
Convolutional	128	3 * 3	ReLU
Max Pool	-	2 * 2	-
Dropout	0.3	-	-
Flatten	-	-	-
Dense	128	-	ReLU
Dropout	0.4	-	-
Dense	10 (Output)	-	SoftMax

Fig. 7 - Iteration 2 Training and Validation Accuracy and Loss



As seen in **Figure 7**, by the time the second model iteration had completed training, it had achieved a training accuracy of 81.9%. After validating the model, it was able to achieve a final accuracy of 91.9%. For this particular model iteration, the number of epochs was increased to 55. Unfortunately, this did not produce the expected result of an increased model accuracy. Although not a significant improvement, our accuracy results indicated two things: that elements of the current iteration model should be included in the next model iteration, and the number of epochs should be decreased. After training, the model's cross entropy loss stood at 0.55. After validating the model, the cross entropy loss stood at 0.29. Our cross entropy loss had decreased in comparison to the initial model iteration, which provided us with another indication that elements of the current model should remain present in the next iteration. As in the first iteration, the learning rate reduction scheduling can be seen following the oscillation patterns seen in the green circles (most evidently, but there are more reductions).

Fig. 8 - Iteration 2 Confusion Matrix



Overall, the confusion matrix for iteration 2 is nearly identical to that of iteration 1. Since our model's accuracy falls within the range of 90-100%, our model was able to correctly identify most digits it was presented with.

7 The Network Architecture

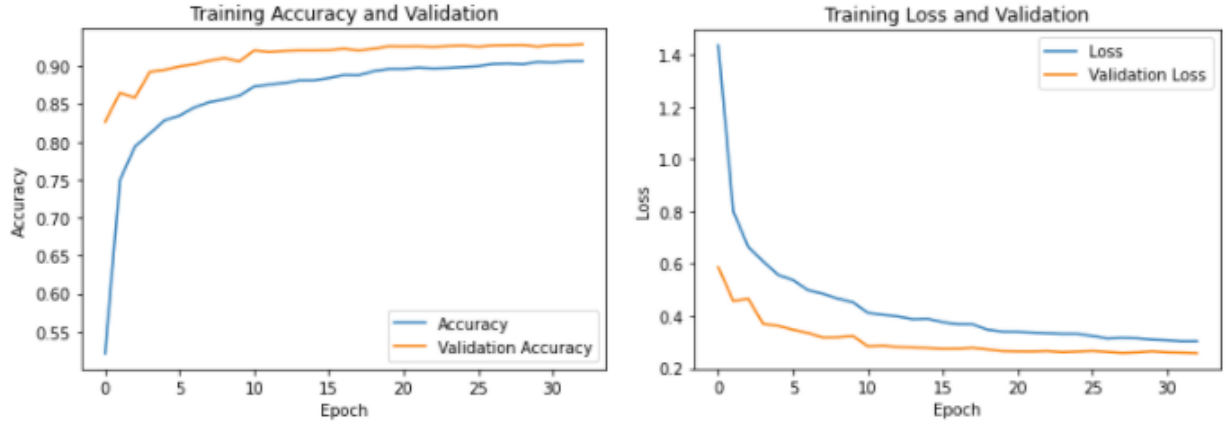
Fig. 9 - Final CNN Architecture

Layer Type	# of Filters / Dropout Rate	Kernel Size	Activation Function
Convolutional	32	3 * 3	ReLU
Max Pool	-	2 * 2	-
Convolutional	64	3 * 3	ReLU
Max Pool	-	2 * 2	-
Convolutional	128	3 * 3	ReLU
Max Pool	-	2 * 2	-
Convolutional	256	3 * 3	ReLU
Flatten	-	-	-
Dense	128	-	ReLU
Dropout	0.3	-	-
Dense	10 (Output)	-	Softmax

The final iteration of our CNN architecture consists of a series of 4 convolutional layers, each followed by a max pooling layer. The initial convolutional layer includes 32 output filters ReLU activation. Each subsequent convolutional layer increases the filter size by a factor of 2. The output of the final convolutional-maxpool pair feeds into a flatten layer that reduces the dimensionality of the input to . The input of the flatten layer is then fed into a dense layer using ReLU activation that further reduces the dimensionality of the output, followed by a dropout layer with a frequency value of 0.3 to prevent overfitting. The final layer in our architecture is another dense layer that uses softmax activation to produce 10 probabilities, the highest of which representing the most likely digit contained within the image. The ImageDataGenerator parameters were reduced to 8 degrees rotation, 10% height variation, 10% shear intensity, and 95%-105% zoom. The batch size was increased dramatically to 250 over the same 100 epochs. Early stopping and learning rate reduction were also set to monitor validation loss with a patience of 5 epoch in early stopping and 2 epochs in learning rate reduction, which featured the original 50% learning rate reduction. The previous graphs show higher oscillation patterns in the validation loss, which could make validation loss a better candidate for what these functions should monitor.

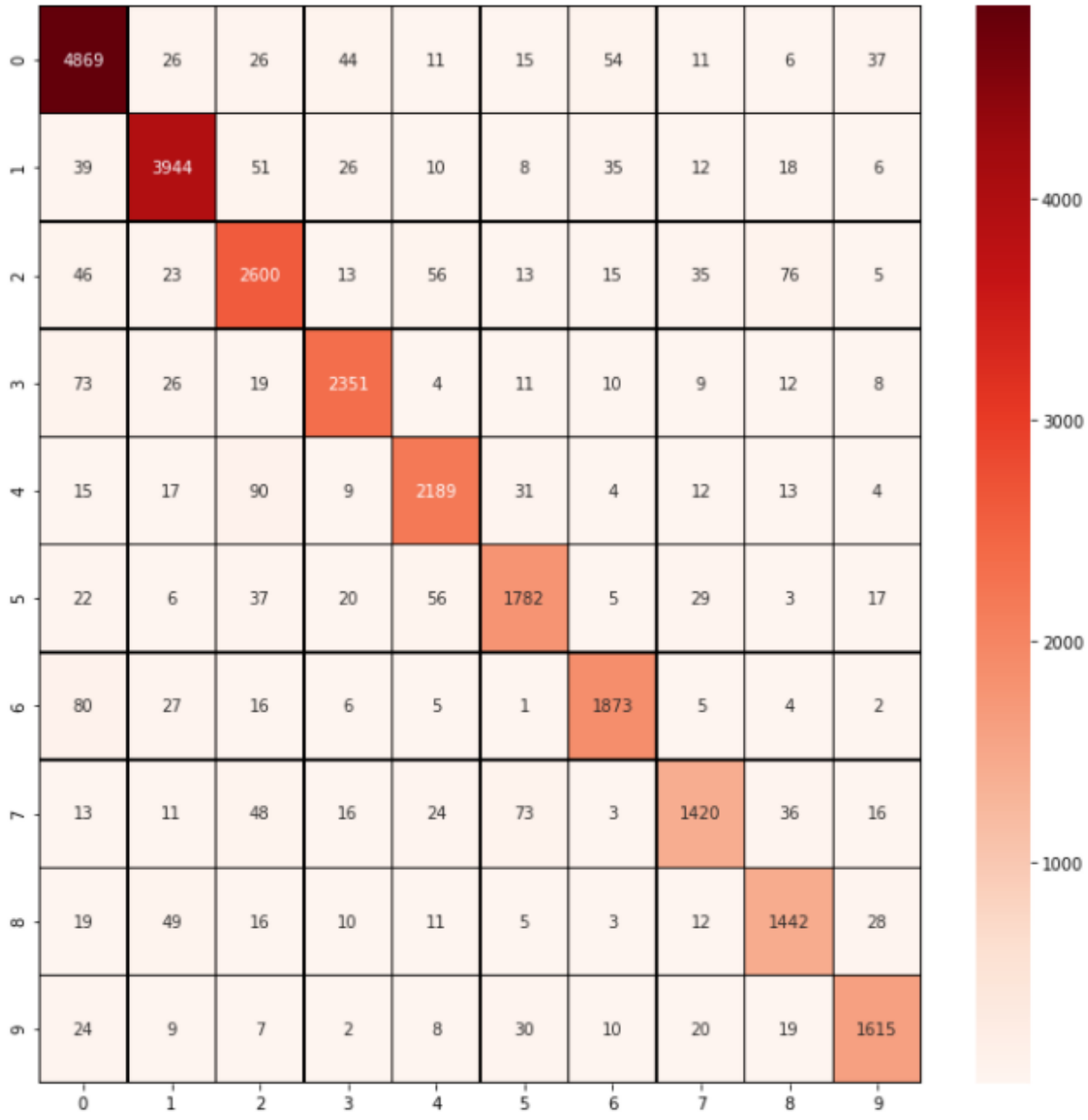
8 Interpretation of Output

Fig. 10 - Final Training and Validation Accuracy and Loss



The final model achieved a training accuracy of 90.6%. After validating the model on the test dataset, it was able to achieve a final accuracy of 92.5%. By both decreasing the number of epochs and modifying our networks architecture, the validation accuracy of the model was increased. Again the effect of learning rate reduction is highly prevalent. After training, the model's cross entropy loss stood at 0.3. After validating the model with the test dataset, the cross entropy loss stood at 0.27. The results produced here ran analogous to the training accuracy and validation results. It seemed as if a combination of tweaking our CNN's architecture and slightly decreasing the number of epochs had a net positive impact on our model's performance. This further decrease in loss could also, at least in part, be attributed to using validation loss in our early stopping and learning rate reduction function as was earlier hypothesized.

Fig. 11 - Final Confusion Matrix



In comparison to the previous two confusion matrices, no drastic changes have taken place. The overall number of correctly identified digits has increased in comparison to previous results.

9 Implementation and Future Work

To create a CNN capable of being implemented into a fully-realized scene text recognition system, the accuracy of said model would have to exceed the accuracy of the final model iteration. Whereas the final model iteration achieved an accuracy of 92.5%, an acceptable CNN would display a validation accuracy in the range of 98-100%.

All CNN iterations were produced using the Keras TensorFlow API. In the future, CNN architectures constructed via alternative Python libraries could be explored. Promising choices include Caffe (a deep learning framework developed by Berkeley Artificial Intelligence Research), Apache's MXNet library (which is compatible with Python, C++, Java, and a host of other programming languages), or Theano (which may provide superior means of controlling and optimizing low-level matrix operations compared to MIT's Keras API).

Should such a CNN be successfully constructed, it would be appended to a hardware system that would feed camera frame data into a text detection/localization algorithm. Once the text is successfully localized, each individual digit would be passed to the CNN. The final output filter would produce a digit readable by some other software system. Applications of a fully realized system would include autonomous vehicle address-finding, automated mail delivery, or automated reading of license plates.

10 Conclusion

This experiment sought to find a CNN layer structure, hyperparameter, and utility function configuration in order to achieve a model accuracy of over 90% on the SVHN 32x32 dataset. It was discovered that dropout layers between the convolutional and max-pooling layers may have had a negative effect on the model. This is evidenced in the progression of having 2 dropout layers in the first model, 4 in the second, and then just one between the 2 dense layers in the last. A simpler pattern of:

Convolutional→MaxPooling→Convolutional→MaxPooling→Convolutional→MaxPooling → Convolutional→MaxPooling→Flatten→Dense→Dropout→Softmax

seemed to work better than incorporating additional batch normalization layers or having convolutional layers next to one another.

Unequivocally, learning rate reduction and early stopping both had a clearly beneficial role in training as evidenced by the accuracy and loss curves. It was discovered, however, that observing these graphs in previous versions can assist in providing clues of what metrics these functions should follow. In this case, it was evident that there was more variability in the validation loss curves, which were ultimately chosen as the monitoring method for them. Likewise, the ImageDataGenerator also assisted in refining loss and increasing accuracy, as it was the major differentiating factor between models 1 and 2. The parameters initially presented to this function were likely too high as there was already a high amount of variability in the images. If any number is rotated too far, it is likely that it may no longer be recognized as the

same number. In this case, it may even have a detrimental effect in training. The dataset may also have been large enough to account for most of this variation as is, but there was likely some benefit in its addition to the dataset.

The AMSGrad variant proposed in the 2018 International Conference on Learning Representations (ICLR 2018) article, *"On the Convergence of Adam and Beyond"*, was also tested for its efficacy in these experiments [3]. Though likely not a tremendous influence, its introduction came before a marked improvement between models 1 and 2. Admittedly it's hard to tell what the exact effect was without isolating its use from other parameter changes.

Model performance seemed greater with a higher number of epochs, especially with the callback functions aiding in the prevention of overtraining. Likewise, larger batch sizes also seemed to have a positive impact on model performance as the first 2 models had batch sizes of 50 and the last a batch size of 250.

There are limitations to the conclusions we can draw from this data, however. Given that models often took upwards of 5+ hours to train, many parameters were changed simultaneously in an attempt to diminish the time needed to train a successful model. This makes it difficult to discern the individual effects of each parameter change. This was attempted to be mitigated by training several different versions in parallel on the same machine, however, this ultimately led to slowed system performance, hardware overheating and eventually the application crashing before it was fully trained. For future iterations and study, it is suggested to break these parameters out into single changes to examine their individual effects. This would be made much more possible by running the models on a high-powered GPU.

11 References

1. Yuval Netzer, Tao Wang, Adam Coates, Alessandro Bissacco, Bo Wu, Andrew Y. Ng Reading Digits in Natural Images with Unsupervised Feature Learning *NIPS Workshop on Deep Learning and Unsupervised Feature Learning 2011*.

2. Wood, T. *Convolutional neural network* https://deepai.org/machine-learning-glossary-and-terms/convolutional-neural-network.

3. Reddi, Sashank J., Satyen Kale, and Sanjiv Kumar. "On the convergence of adam and beyond." *arXiv preprint arXiv:1904.09237* (2019).

4. "API Reference." *Scikit Learn*, scikit-learn.org/stable/modules/classes.html.

5. *Numpy and Scipy Documentation*, docs.scipy.org/doc/.

6. "TensorFlow Core v2.4.1 - API Documentation." *TensorFlow*, 5 May 2021, www.tensorflow.org/api_docs/python/tf.