Nearly Outstanding Character Recognition Model (NOCRA)

Noah Baxley
nbaxley@hawk.iit.edu

Lucas Ferguson
lferguson@hawk.iit.edu

Astrid Beasley
nbeasley@hawk.iit.edu

Fatih Cetin
fcetin@hawk.iit.edu

Vishvam Patel
vpatel138@hawk.iit.edu

***Abstract*—This paper presents the development of a Convolutional Neural Network for optical character recognition focused on identifying single handwritten alphanumeric characters. The system employs a feedforward neural network trained on the EMNIST [1] dataset. The model architecture includes multiple hidden layers employing various techniques to ensure an efficient, accurate output that converges quickly. With many iterations of testing and tailored network design, we were able to create a network that achieves a recognition accuracy of 90%. This work aims to serve as a foundational framework for OCR applications and a shining example of the applications of convolutional neural networks.**

Keywords—CNN, optical character recognition, MNIST, EMNIST

I. Introduction

The objective of this assignment was to undertake a project within the field of machine learning, providing practical experience in addressing a real-world problem. For this purpose, we selected the task of optical character recognition (OCR) on handwritten characters, a well-established application in machine learning. Convolutional Neural Networks (CNNs) were chosen as the primary method due to their proven effectiveness in OCR tasks and relative ease of implementation. Preliminary experiments revealed that most variants of the CNN architecture performed well on this task, so we decided to take a different approach and add a dimension to this assignment. The added dimension: research and experiment with different methods to find which ones influence the performance of a CNN.

The findings of this project demonstrated that the performance of a basic CNN could be significantly enhanced through several key techniques. These included the implementation of batch normalization, the use of a stochastic gradient descent optimizer, and the application of smaller max pooling kernels. Additionally, experimental evaluation identified an optimal configuration of approximately 200 neurons in the first classification layer.

II. Related Work

The first example of character recognition is LeNet created in 1989 which just aimed to recognize handwritten digits from zero to nine. LeNet 5 [2] was published in 1998, and its architecture consisted of two convolutional layers in the feature recognition layer and two dense layers connected to one output layer in the classification layer. This model performed well for the time, but there have been many new methods that have come out since then.

We also found several models published online that were very helpful in our experimentation.

A Jupyter notebook [3] submitted to a contest in 2023 by Mohammad Shahid attempted the same task as LeNet of recognizing digits 0 to 9. His model consisted of two convolutional layers and just one dense layer connected to one output layer compiled with Keras's Adam optimizer. He was able to achieve a best score of 98% accuracy with his model during that contest.

We found a repository created earlier this year [4] by a team for their own school project. They attempted to create a model that classified the Arabic Handwritten Character Dataset (AHCD) which has 28 classes. The model was much more complex and had 8 convolutional layers, 4 max pooling layers, and 2 dense layers leading into an output layer.

III. Implementation

Our implementation uses Keras's CNN. The final model has two *Conv2D* layers, both using a rectified linear unit (ReLU) activation function. The first layer has 8 kernels of size 5 and the second has 16 kernels also of size 5. After both is a *MaxPool2D* layer with kernel size and stride of 2 by 2. Added between them is a *BatchNormalization* layer. After the feature extraction layer is a *Flatten* layer which outputs into a *Dense* layer. After experimenting with multiple values, we landed on using 200 neurons here with a ReLU activation function. For training purposes, we added a *Dropout* layer between this dense layer and the output which was another *Dense* layer of 47 neurons using a softmax activation function to generate outputs.

For training, we used images from the Extended MNIST (EMNIST) dataset [1] EMNIST was created by The MARCS Institute for Brain, Behavior and Development to expand the Modified National Institute of Standards and Technology (MNIST) [5] dataset by adding images from NIST. They also categorize the data by five metrics: *page*, *author*, *field*, *class*, and *merge*. For our purposes, we chose the *merge* hierarchy which groups characters by how similar they are. As an example, the difference between "C" and "c" is hard to detect if you don't have some sort of reference, so the *merge* hierarchy groups

letters that follow that pattern: C, I, J, K, L, M, O P, S, U, V, W, X, Y and Z are all combined with their lower-case counterparts creating a new dataset with 47 classes instead of 62. There are just over 810,000 images in EMNIST, and each is 28 by 28 pixels.

For training, we used an 80/20 test to train ratio and trained each model for 25 epochs.

IV. Experimental Evaluation

For the demo set that we made available, 50 items from each label were selected (2350 images total) and made available on OneDrive. During training, we used all images, and we split the images randomly into 80% test 20% train.

To utilize those images during training, we put each class of image into its own directory. Windows does not distinguish between uppercase and lowercase text however, so to get around this we prefixed each class with a *lower_* or *upper_* to distinguish between the two. We used one hot encoding to distinguish images while training, so these label prefixes did not affect any of the training.

In total we made 9 significant checkpoints in our process which we associated with 9 different models that we choose to present in this paper. Each checkpoint represents researching different options and testing the effectiveness of each change.

To make it more clear how the model progresses at each checkpoint. An area under the curve (AUC) metric was used. The logic being that a model that performs better will converge faster and will have a higher AUC value.

The first and second models are uninteresting, and they were simply the first successful attempts that worked. There were issues we ran into with trying to read images into TensorFlow, labeling them correctly, and shuffling them correctly for training that we had issues with at first. The way we were labeling images also is a bit unorthodox for projects like this, so it was hard to find help online. For a while we could only get just digits to work correctly. The results of models 1 and 2 have passable performance when performing on the whole dataset.

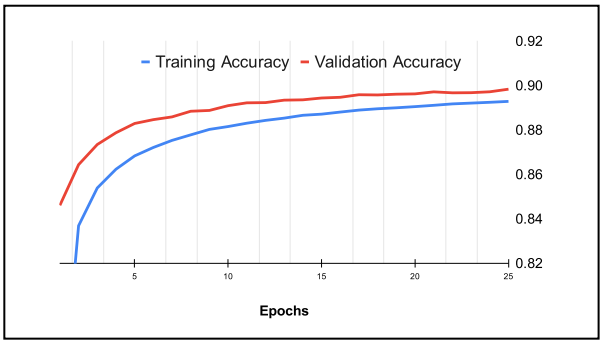

Fig. 1. Training and validation accuracy over of model 2 over 25 epochs. Achieved a maximum validation accuracy of 0.898 and an AUC of 21.34.

The third model we created used a stochastic gradient descent (SGD) optimizer as opposed to the Adam optimizer we had used in models 1 and 2. The optimizer used when training a neural network describes how the weights will update to minimize loss. While both Adam and SGD use a gradient descent of some type, the main difference is the way that Adam adaptively scales the learning rate it uses based on history. Simply out of curiosity because we learned it in class, we tried using SGD instead and noticed a consistent improvement across multiple tests when it came to validation accuracy. This was a bit counterintuitive considering Adam is designed to perform better. Looking around though, we found an article [6] that demonstrated code and that pointed us to a paper [7] that also observed these results. Granted the differences were small (the final difference in validation accuracy between model 2 and model 3 was less than 0.01 after 25 epochs, but the SGD optimizer converged quicker than Adam.

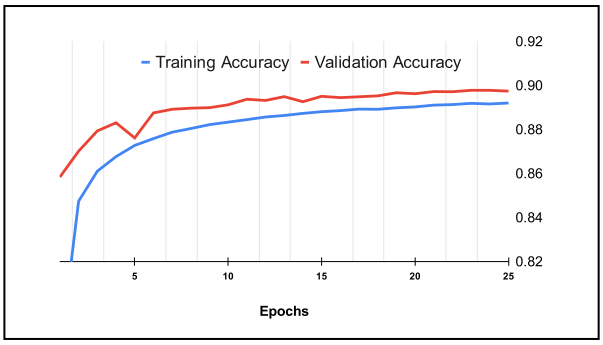

Fig. 2. Training and validation accuracy over of model 3 over 25 epochs. Achieved a maximum validation accuracy of 0.897 and an AUC of 21.37.

The fourth model experimented with increasing the size of the kernels and strides used for the *MaxPooling2D* layers and the first *Conv2D* layer. We noticed a comparatively large drop in performance as soon as we implemented this. We believe that this may have been related to the fact that we were using such small images. Regardless, we quickly abandoned the idea of increasing the kernel sizes.

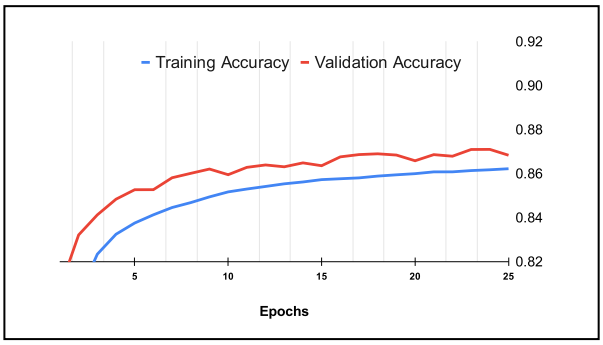

Fig. 3. Training and validation accuracy over of model 4 over 25 epochs. Achieved a maximum validation accuracy of 0.868 and an AUC of 20.64.

In the fifth model, there were not any significant structural changes, but we did make changes to the way we were currently working. Prior to this point we were working with a smaller subset of EMNIST that someone had published that used 128 by 128 images. At this point we switched to using the full EMNIST dataset which consisted of just 28 by 28 images. To speed up training we also changed the way that TensorFlow processed images. Noah Baxley had a computer with 32 gigabytes of memory, and we were already using his computer to train because he had an RTX 3060, so instead of caching images to the file system we opted for just caching them in RAM which

by itself sped up training significantly. We also started using TensorFlow's data API [11] to optimize the batch size. While there were no significant model changes at this step, it was a meaningful change in our workflow.

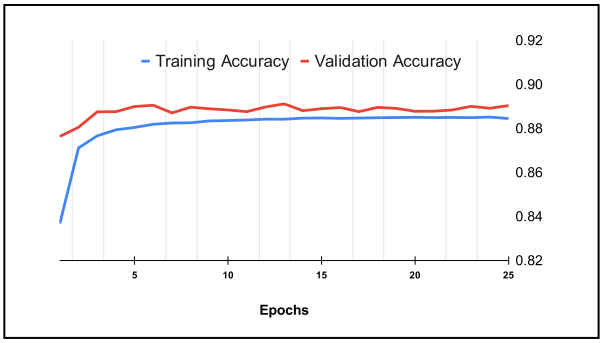

Fig. 4. Training and validation accuracy over of model 5 over 25 epochs. Achieved a maximum validation accuracy of 0.890 and an AUC of 21.31.

In the sixth model, we added in BatchNormalization layers. Batch normalization standardizes the inputs to a layer for each batch during training which can have a regularizing effect on training and prevents outliers from having too much of an effect as well. Overall, it makes the gradient descent smoother. Consequently, because the descent is smoother, the optimizer can perform better, and the model converges faster. We were pleasantly surprised by how effective this change was on performance.

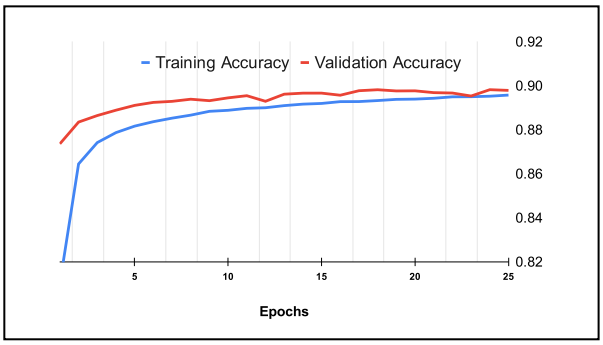

Fig. 5. Training and validation accuracy over of model 6 over 25 epochs. Achieved a maximum validation accuracy of 0.897 and an AUC of 21.45.

In the seventh model, we tested adding more neurons to the dense layer at the start of classification going from 120 to 200 neurons. Interestingly, this had little effect on the convergence of validation accuracy but instead mainly increased the training accuracy. In the eight model we did the same thing again increasing the neuron count from 200 to 300. When we did that, however, we noticed an increase in training accuracy above validation accuracy. That was an indicator that our model might be overfitting, so we decided to reduce the neuron count.

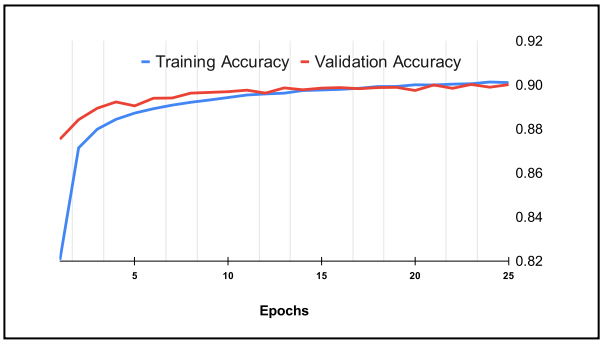

Fig. 6. Training and validation accuracy over of model 7 over 25 epochs. Achieved a maximum validation accuracy of 0.900 and an AUC of 21.50.

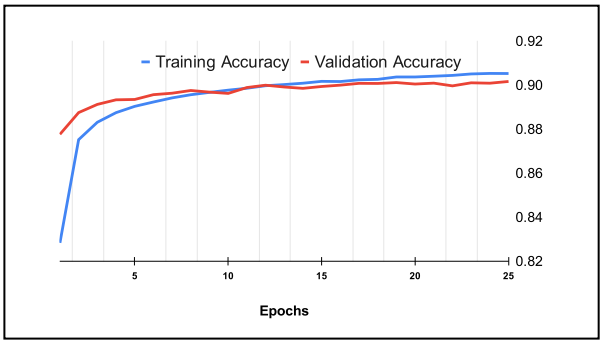

Fig. 7. Training and validation accuracy over of model 8 over 25 epochs. Achieved a maximum validation accuracy of 0.901 and an AUC of 21.54.

For the ninth and final model, we reduced the neuron count back to 200 and moved the *BatchNormalization* layers from being after the activation functions in the convolutional layers to before them. This moving did not seem to have that large of an effect though.

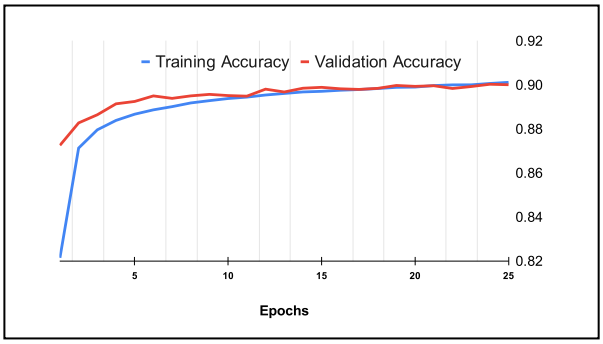

Fig. 8. Training and validation accuracy over of model 9 over 25 epochs. Achieved a maximum validation accuracy of 0.900 and an AUC of 21.50.

V. Results

Our final model achieved a maximum validation accuracy of 90% and we were satisfied that all our changes were able to produce such great results. When looking at an overlay of all the data comparatively, it does not perform better than the model that may have been overfitting (model 8), but excluding that, it performs the best with a final accuracy of 0.900 and an AUC of 21.50.

Figure 1 shows an overlay of the validation loss reported at each epoch during training and figure 2 shows the reported validation accuracy.

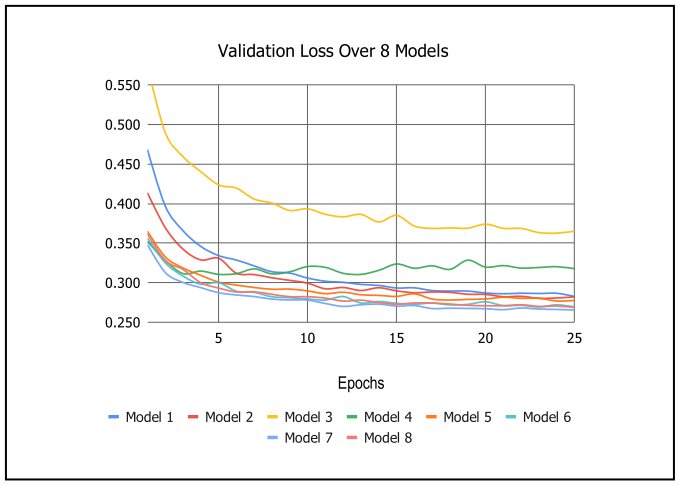

Fig. 9. Comparison of validation loss over the course of 25 epochs for our models.

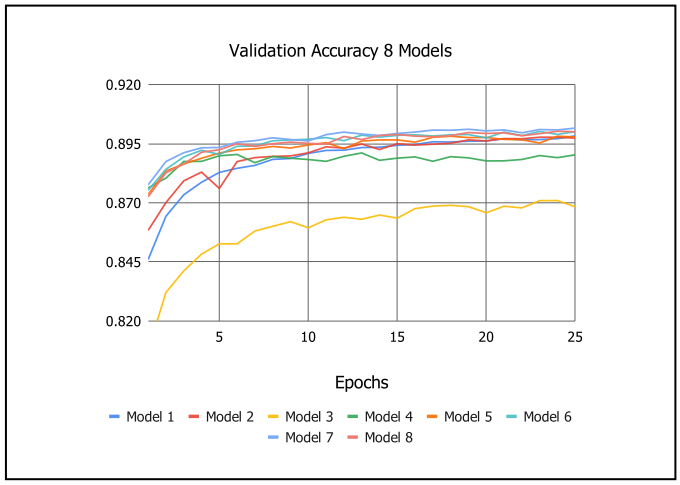

Fig. 10. Comparison of validation accuracy over the course of 25 epochs for our models.

As the accuracy graph shows, all our higher achieving plateau at around 90% validation accuracy. If we had the chance, we would like to explore this further.

When comparing AUCs at each checkpoint, the improvement become more evident. There is a clear difference in performance between the first model and the last. The lack of performance by the fourth model can also clearly be seen.

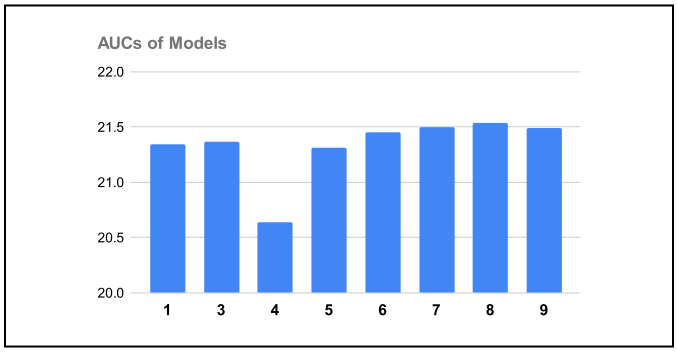

Fig. 11. AUCs of model performance at the different checkpoints.

VI. Future Applications

One possible application of this research is the enhancement of note-taking systems like Excalidraw, a popular tool for visual thinking and digital whiteboarding. By incorporating an OCR system, users could seamlessly convert handwritten words within Excalidraw canvases into easily editable and searchable text boxes. This feature would improve the efficiency of organizing and retrieving information, making tools like Excalidraw even more versatile for students, professionals, and educators.

VII. Conclusion

Our model performed well; most architectures produced a maximum accuracy of around 90% on the validation data. We believe that our approaches to decrease the time to convergence of batch normalization and using stochastic gradient descent were a key factor in reaching that.

Despite the promising outcomes we achieved, the model appears to have reached an upper accuracy bound. Future work could continue improving by investigating the causes of this limitation. We could also benefit from further exploration of more advanced techniques that might not be commonly used in research found online.

Here we demonstrate the potential of Convolutional Neural Networks for optical character recognition and serve as a foundation for further research into efficient and scalable OCR projects.

References

[1] Co en, Gregory, et al. "EMNIST: Extending MNIST to handwritten letters." Proceedings of the International Joint Conference on Neural Networks (IJCNN), 2017, pp. 2921-2926, https://ieeexplore.ieee.org/document/7966217.

[2] LeCun, Yann, et al. *"Gradient-based learning applied to document recognition."* Proceedings of the IEEE, vol. 86, no. 11, 1998, pp. 2278-2324, https://ieeexplore.ieee.org/document/726791

[3] Shahid, Mohammad. *"7-CNN Handwritten Digit Recognition."* Kaggle, 2023, https://www.kaggle.com/code/itsmohammadshahid/7-cnn-handwritten-digit-recognition

[4] Faris771. *"Handwritten Character Recognition."* GitHub, 2024, https://github.com/faris771/Handwritten_Charactar_Recognition.

[5] LeCun, Yann, et al. *"The MNIST Database of Handwritten Digits."* http://yann.lecun.com/exdb/mnist/

[6] Shaoanlu. *"SGD, All Which One is the Best Optimizer? Dogs vs. Cats Toy Experiment."* Shaoanlu's Blog, 29 May 2017, https://shaoanlu.wordpress.com/2017/05/29/sgd-all-which-one-is-the-best-optimizer-dogs-vs-cats-toy-experiment/.

[7] Ashia C. Wilson, Rebecca Roelofs, Mitchell Stern, Nathan Srebro, and Benjamin Recht. 2017. The marginal value of adaptive gradient methods in machine learning, https://dl.acm.org/doi/10.5555/3294996.3295170.