Handwritten Symbol Classification with Pre-Trained Convolutional Neural Network

Dylan Chung, Matthew Reinhard, Jackson Kaprak

***Abstract*—Despite advancements in the performance of machine learning models with application to image classification, the task of handwritten symbol recognition poses a few unique challenges. With respect to our dataset of handwritten math symbols, difficulties include a variety of handwriting styles, the presence of background noise (lines in the paper and different shading), similar-looking classes, photographic angle, and incorrect labeling.**

We explored several methodologies to account for these challenges, performing external research and first hand testing to guide model selection. After conducting preliminary research and testing, we concluded that convolutional neural networks were the most effective at accomplishing our task. By utilizing data augmentation techniques in order to increase the input space and generalizability of our model, we were able to achieve favorable results on unseen test data. By combining these techniques with the leveraging of transfer learning through pre-trained models like resnet50, we were able to attain high levels of accuracy on unseen data.

This report will outline our review of the relevant literature, details surrounding the implementation of our model, and a discussion of our experimental procedure.

I. Introduction/Literature Review

We explored several overarching model types to accomplish our task, specifically support vector machines, multi layer perceptrons, and convolutional neural networks (further referred to as SVM, MLP, and CNN respectively). Soft-margin SVMs initially appeared promising due to their inherent robustness to outliers and ability to handle high dimensionality. Our initial research supported this intuition, with one SVM based model achieving 93% accuracy on an image classification task, albeit one with a small dataset [1]. However, even after testing with a combination of StandardScaler and PCA, performance on our project's training and unseen data was inadequate. The highest accuracy score we reported on test data after hyperparemeter tuning was 55%. MLP performance was similarly low, with our initial testing on unseen data hovering around accuracy scores of 50%.

Despite the poor performance of the MLP models, we still decided to explore more advanced neural networks due to the compendium of research supporting neural networks for image classification. Research indicates that CNNs are generally the most accurate with regards to handwritten image classification [2], surpassing SVMs and MLPs on unseen data across all relevant performance metrics. Several pieces of literature that we reviewed were based on the assumption that CNNs are the most effective models at image classification and aimed to maximize performance through testing of different hyperparameters and architectures [3], [4]. The architecture of a CNN is particularly well suited for image classification because it learns the best filters to map the features of the input dataset, which means very little manual hard coding is required [5] .

Therefore, we decided to test the validity of CNNs on our dataset. Our initial test of a CNN model was promising, with a test set accuracy of 67% with just the first blend of hyperparameters. After testing different optimizers (Adam, SGD, AdamW) and different hyperparameters (learning rate, momentum, batch size), our model was still unable to perform higher than 76%. A common thread amongst all of our previous testing was a tendency for the model to perform markedly better on the training data versus unseen test data, indicating overfitting.

Research indicated that data augmentation could be a solution to the problem of overfitting. One paper specified that, "big data driven mode based deep CNNs still have the 'overfitting' problem; that is, the neural network can perform well on the training set but cannot be effectively generalized on the unseen test data" [6]. This paper discussed how data augmentation could be used to expand and diversify the input data. By doing this, models are able to generalize more. After performing extensive data augmentation, performance was raised to 83% after training on 500 epochs.

During our research of CNNs, transfer learning was discussed as a prominent feature that could achieve very high performance. Transfer learning is a method in which information from previous tasks are used to help guide performance for a new, but related, task [6]. Utilizing transfer learning through a pre-trained model enables users to bypass the need of training a neural network for several weeks, as it has already explored a massive feature space similar to the one that you are looking to learn [5].

To further augment performance, we investigated applications of transfer learning to an image classification task. One paper argued that, "It is recommended to use [a] model developed and trained for a task as a starting point of the task that is similar to the trained one" [7]. This research found that a popular pre-trained CNN, resnet50, outperformed several other deep learning models with regards to image classification. Additional research further solidified this claim, with a resnet50 model outperforming even more pre-trained models [5] We therefore decided to begin our exploration of transfer learning with the resnet50 model.

II. Implementation

Like all pre-trained models, resnet50 offers advantages over using a manually built architecture. First, it can leverage multitudes of related information when initializing its weights, allowing for faster training and convergence. In effect, pre-trained models generalize an image classifier on a massive input dataset that is related, but distinct, from your input data. Resnet50 specifically was trained on a dataset of over 14 million images [8]. Furthermore, its architecture was built and optimized to perform well with regards to image classification specifically.

Resnet50 is one of the most commonly utilized architectures of a CNN. It consists of 48 convolutional layers, 1 max pooling layer, and 1 average pooling layer [5]. These are roughly outlined in the figure shown. The first convolutional layer takes the input image and down samples the image by a factor of 2 in both width and height. Then, the data passes through a batch normalization and relu activation layer before being down sampled by another factor of 2 with a max pooling layer [5]. This is aimed at reducing the spatial dimensionality of the images while retaining the important characteristics. These layers comprise resnet50's built in feature extraction.

The resulting data is fed into the next block of convolutional layers; each block is followed by batch normalization and a relu activation layer. This repeats for four convolutional blocks, allowing for the model to learn deep non-linear features [9]. The output of the last relu layer is then passed to the average pooling layer, which further downsizes the feature maps that the previous convolutional blocks extracted. Finally, this layer's output is sent to the fully connected layer. The fully connected layer consists of $n$neurons, with n representing the number of classes. This final layer takes the features generated by the rest of the model and passes them through the softmax activation function for classification [8].

We had to make one alteration to our data because resnet50 and all the other pre-trained models that we tested (resnet18 and densenet), required 3 input channels as they were trained on RGB images. Our data was gray scale, and therefore only had 1 input channel. This initially concerned us, as this may negatively impact our model performance due to the pre-trained model's lack of exposure to gray scale images. We transformed the 1 input channel into 3 equivalent

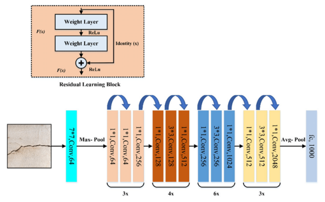

Fig. 1. Resnet50 Architecture

ones in order to be functional with the pre-trained model. Our concerns appear somewhat valid, as we have been unable to replicate the >97 percent accuracy that resnet50 was able to consistently achieve in various settings [5][4][7]. However, we were still pleased with its overall performance.

In terms of implementing the model, it is recommended to use our model with access to GPUs due to the large feature space, the expensive training associated with a 50 layer convolutional neural network, and data augmentation. The model must be run on a kernel with PyTorch (with HiPerGator, we utilized PyTorch 2.2). The input data for both the training and test is expected to be in .npy format. The required packages include PyTorch, PIL, and numpy.

III. Experiments

The first issue we noticed when beginning this project was the lack of consistency in the data between images taken by different students. Initially, we thought that by utilizing data augmentation, we could produce a more uniform data set for training our model. However, we quickly realized that this would be a time consuming and tedious approach. There is no one-solution-fits-all fix to remove outliers and enhance the data at the same time without supervision, despite exploring options such as utilizing a fast Fourier transformation and masking to remove background lines from lined paper [10], and implementing a threshold to isolate the writing within the image dependent on the average pixel value of the image.

After some brief experimentation with SVMs and MLPs, which did not yield any promising results, we focused our attention on building a CNN architecture. For our first attempt at a CNN architecture we included two convolutional layers and three fully connected layer which utilized ReLU activation functions. This was modeled after a similar architecture we found as an example of using PyTorch to train a classifier on the CIFAR10 dataset. While validation set accuracy for this architecture was a significant improvement (75% accuracy) over our previous attempts, our model still did not produce stellar results. We tried adjusting several hyperparameters, such as the number of convolution filters, the number of output features of our fully connected layers, and the number of fully connected layers. We also tried different learning rates, dropout, and different learning algorithms such as Stochastic Gradient Descent, and Adam. Still, our model's accuracy on the validation set did not significantly improve. We began to suspect that we would need to train our model on more data to avoid overfitting. We determined than the technique of data augmentation may resolve our issues.

Data augmentation has many merits. There are a wide variety of augmentations that can be performed on images, allowing for extensive testing and experimentation to determine the transformations that result in a trained model with the highest accuracy in classifying unseen images. Since our test data is taken from the overall collected data, it likely contains the same issues we have with the training data, and so training with data augmentation will not hurt our model that much. To top this off, there are also methods that have already been developed, which automatically augment the data in specific ways that have been tested and shown to help image classifiers produce better testing results [11], [12], [13]. Given this fact, as we found that our model suffered from significant overfitting in our initial testing, data augmentation became a main focus in our experimentation and implementation.

By the time that we began our experimentation with data augmentation, we had already decided on utilizing a CNN for our base model, since it had the best baseline between the three models we initially tested. We decided to focus on accuracy score as our sole metric to focus on, as that is what our model will be evaluated with. We considered evaluating f1 scores as well, but decided it would be less effective due to the equal distribution of input classes. We also had a maximum test value of 77% at this point. Initially, we hypothesized that by either augmenting the existing data, or adding augmented data on top of the original data to train with, that we could reduce overfitting and increase our test value by upwards of 10-15% [13]. Doing some quick research resulted in us finding the pytorch library called transforms and transforms.v2, which were specifically designed to provide users with a library of transformations for use whilst training an image classifier or modifying an existing image. We were already utilizing this library to convert our data set from a .numpy array to a tensor which was used by the CNN during training and validation, along with a normalization with a mean and standard deviation of 0.5, based on the CIFAR10 training set.

To add any of the different transformations we wanted to test on our data, all we had to do was introduce the transform into the pipeline of transforms that already existed. Then, when loading our dataset, the transforms would automatically be applied to the data, and we could train and validate using the altered data. Ultimately, we did not see much if any performance uplift from our various testing through this methodology: at most a gain of 1% for a test accuracy of 78%. We realized from this that since we were doing training with validation, that if we wanted to validate our results using images close to what would be seen in our testing dataset, that transforming all of the training data before the training validation split means the data used for validation would be affected by the data augmentations, and not accurately represent real world data (our test dataset). Thus we resolved to retool the transformation pipeline to augment only the training images after we separate these from the validation set. This meant that whilst training would take longer, our validation would be much closer to what we expect the model to classify during testing.

For this next experimental run, we decided to change a few things from our initial testing. We allowed our training to run for an increased duration of 500 epochs (from 200 epochs, and removed the patience interval), and changed out the automatic augmentation method for the random augmentation method (which we left at default values: num_ops = 2 and magnitude = 9). We decided to try the random augmentation since we saw it could produce results better than the automatic augmentation could [13]. We also thought it also would require a more extensive feature search since the method random chooses different augmentations to use each time, thus producing varying results per iteration. Ultimately, this resulted in an increase to our test accuracy to 83%, despite a grueling computation time and utilizing GPUs.

Next we tried using the AutoAugment function from the torchvision library to perform our data augmentation. The AutoAugment function has 3 different ”policies” which designate how the input images will be augmented. The three policies are IMAGENET, CIFAR10, and SVHN. We first applied only the IMAGENET policy in augmenting our data, and combined this with our original training dataset to train our model. We experimented with allowing the training to run for a larger number of epochs, adjusted the learning rate, and added significantly more (augmented) training data.

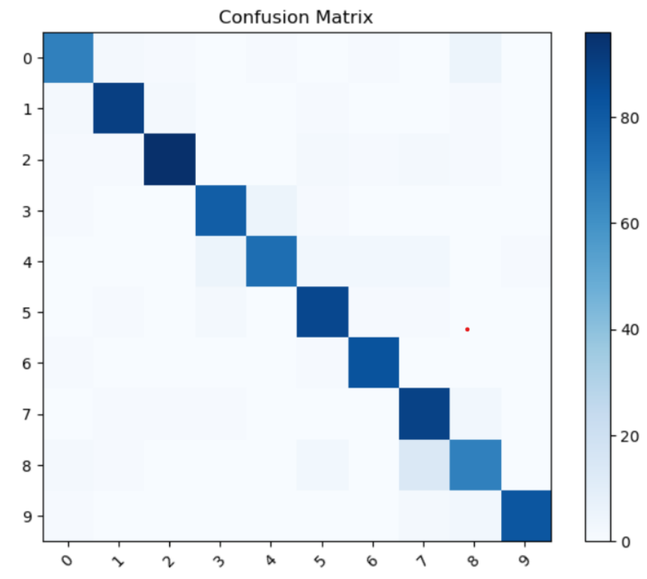

Fig. 2. Confusion Matrix, Final Model

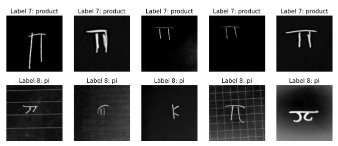

Fig. 3. Product vs Pi

Ultimately, we did not find significant improvement on our validation set accuracy score with this method. Next we tried all three of the augmentation policies. We combined these three augmented data sets with our original training data and found significant improvement for our image classifier. Our final model had a 92% accuracy on the validation set. When viewing the the confusion matrix of our model on the validation set, we see that the most common misclassification by far is between class 7 and 8, which represent the pi and product symbols. This elevated level of misclassification most likely occurred due to the visual similarity of these two images. As seen in figure 3, several hard to distinguish examples exist even among a small random sampling of images from these two classes. We attempted to improve upon this 92 percent performance by tuning hyperparameters, but were unable to achieve any improvement. For optimizers, we found that Adam slightly outperformed SGD and AdamW when comparing across batch size and learning rate, with neither optimizer able to surpass an 89 percent accuracy score. A learning rate of 0.001 appeared to be optimal for Adam, as higher learning rates decreased performance but higher ones had no impact and only increased computational complexity.

IV. Conclusion

‘ There were several challenges associated with classifying handwritten math symbols; ranging from misclassification to background noise to similar looking symbols.. To begin the project, we focused on finding methods which could provide more consistency within the input data, such as Fourier transformation and masking, in order to combat these challenges. We quickly realized that these methods would be computationally expensive and would not effectively accomplish what we were seeking.

We next sought to classify handwritten symbols with SVM and multi layer perceptron models, Even after performing dimensionality reduction and a grid search for hyperparameter tuning, our models were computationally expensive and continued to perform poorly on both validation and test data.

Research indicated that CNNs would be a more effective method. Our initial testing appeared to confirm the literature. However, our models tended to over fit as we increased the number of epochs. To combat this tendency, we tested various data augmentation techniques and leveraged transfer learning to improve the generalizability of our model. We decided to use resnet50 architecture with cross entropy loss and the adam optimizer. This combination led to significantly higher performance, achieving a 92 percent accuracy score on the validation set.

Completing this project gave us a few insights. We learned that pre-trained models can serve as a great starting point for classification tasks, and are more versatile than initially thought. Resnet50 performed well on a dataset of gray scale images, despite being optimized for RGB images. The importance of limiting computational complexity was further reinforced; training our early models on local machines was extremely time consuming and inefficient, and more advanced models had long training times even when utilizing GPUs. Overall, we were pleased with the performance of the model.

References

[1] S. Chaganti, I. Nanda, K. Pandi, T. Prudhvith, and N. Kumar, “Image classification using svm and cnn,” 03 2020, pp. 1–5.

[2] R. Dixit, R. Kushwah, and S. Pashine, “Handwritten digit recognition using machine and deep learning algorithms,” *International Journal of Computer Applications*, vol. 176, no. 42, p. 27–33, Jul. 2020. [Online]. Available: http://dx.doi.org/10.5120/ijca2020920550

[3] M. Jain, G. Kaur, M. Quamar, and H. Gupta, “Handwritten digit recognition using cnn,” 02 2021, pp. 211–215.

[4] F. Chen, N. Chen, H. Mao, and H. Hu, “Assessing four neural networks on handwritten digit recognition dataset (mnist),” 2019.

[5] S. Mascarenhas and M. Agarwal, “A comparison between vgg16, vgg19 and resnet50 architecture frameworks for image classification,” *2021 International Conference on Disruptive Technologies for Multi-Disciplinary Research and Applications (CENTCON)*, 11 2021. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/9687944

[6] Z. Qinghe, M. Yang, X. Tian, N. Jiang, and D. Wang, “A full stage data augmentation method in deep convolutional neural network for natural image classification,” *Discrete Dynamics in Nature and Society*, vol. 2020, pp. 1–11, 01 2020.

[7] A. Shabbir, N. Ali, A. Jameel, B. Zafar, A. Rasheed, M. Sajid, A. Ahmed, and S. Dar, “Satellite and scene image classification based on transfer learning and fine tuning of resnet50,” *Mathematical Problems in Engineering*, vol. 2021, 07 2021.

[8] N. Kundu, “Exploring resnet50: An in-depth look at model architecture and code implementation,” 1 2023.

[9] T. Sharma, “Detailed explanation of resnet cnn model.” [Online]. Available: https://medium.com/@sharma.tanish096/detailed-explanation-of-residual-network-resnet50-cnn-model-106e0ab9fa9e

[10] V. Zagustin, “Removing lines from ruled paper with fourier transform.” *GitHub*, 05 2015. [Online]. Available: https://vzaguskin.github.io/fourielines1/

[11] E. D. C. , B. Zoph, D. Mane, V. Vasudevan, and Q. V. Le, “Autoaugment: Learning augmentation strategies from data,” *Google Brain Residency Program*, 04 2019. [Online]. Available: https://arxiv.org/pdf/1805.09501.pdf

[12] K. Tokarski and K. Łecki, “Why automatic augmentation matters,” *NVidia Developer Technical Blog*, 05 2023. [Online]. Available: https://developer.nvidia.com/blog/why-automatic-augmentation-matters/

[13] S. Raschka, “Comparing different automatic image augmentation methods in pytorch,” *Ahead Of AI*, 01 2023. [Online]. Available: https://sebastianraschka.com/blog/2023/data-augmentation-pytorch.html