# Using a Genetic Algorithm to Optimize a CNN for Character Recognition

Andria Legon1

1Department of Modeling, Simulation and Visualization Engineering, Old Dominion University, Norfolk, VA

*Abstract*— **Convolutional Neural Networks (CNNs) are an important machine learning tool for analyzing visual images. Since applications for these networks vary widely, so do the appropriate CNN parameters. Finding the correct parameters is a time-consuming process. This paper examines the effectiveness of applying a genetic algorithm to automatically learn the optimal parameters of an already defined CNN structure.**

**A CNN is designed by hand and its structure is used as the base for the genetic optimization. The genetic algorithm starts by initializing a set of random individuals. In each generation, a set of genetic operations remove weaker individuals, based on classification accuracy, and build stronger ones. The genetic algorithm is tested against the MNIST character set, a standard image classification dataset. The efficacy of the methods used in this paper can be directly compared to other approaches. The genetic algorithm improved classification accuracy from 99.59% on the hand-tuned control to 99.67%. Visualization of the designed network shows a final output layer with well-defined characteristics and little noise.**

*Keywords*—**Convolutional Neural Network, CNN, genetic algorithm, keras-vis, keras**

### I. Introduction

Convolutional neural networks require significant computing power to train. With advancements in computing power, more complex networks are achievable. Today, the average convolutional neural network has numerous features that need to be selected. The number of permutations are unmanageable. Each configuration requires significant effort in training and evaluating its performance. Very quickly, the question of how one efficiently determines the best CNN for a particular problem arises.

One solution involves hand optimizing the CNN. A series of experiments are conducted in which features are added, subtracted or changed one at a time. Results of prior experiments determine selection of new modifications. Each change requires retraining of the network and subsequent performance evaluation. Eventually, the optimal CNN is discovered. Hand optimizing is labor intensive with the designer heavily involved at every step. Its efficacy is highly dependent on the skill of the designer.

Automated solutions through brute force algorithms require no human intervention. The algorithm systematically creates and evaluates every permutation of the possible feature values. Brute force is computationally expensive. There is no logic incorporated to reject features that consistently reduce network performance. Every feature, regardless of performance, are combined with every other feature until all combinations are evaluated.

Another option, discussed in this paper, is the use of a genetic algorithm to optimize the CNN. Exploring genetic algorithms to optimize neural network parameters started in the early 1990’s [1, 2]. It is only in the last few years that genetic optimization of CNNs have been studied therefore there is not a large body of work describing this approach. In [3] the author describes an investigation similar to this paper of optimizing a pre-defined CNN structure. In [4], the authors describe their technique of optimizing a CNN structure.

Given a set of input parameters, genetic algorithms search for the optimal solution to a problem. Genetic algorithms are modeled after the biological evolution seen in nature. The fittest specimens survive to produce new individuals and the poorer entities are removed from the genetic pool. The occasional mutation ensures that the algorithm does not find itself converging onto a less than optimal evolutionary path with no way to escape. The genetic algorithm approach reduces the human workload required for hand optimization. Unlike brute force algorithms, the genetic algorithm will remove non-performing features from its feature set reducing the number of evaluations required to find an optimal CNN.

Visualization of the optimal network helps us more intuitively grasp the processing of the layers and state of the network. For the output layer, a visualization shows whether or not the network is well-trained. The ideal image that activates a particular classifier should have hints of the classified image in them. If the visualized image has no discernable patterns, it points to a network that may not be sufficiently trained regardless of accuracy rates.

### Method

## Hand Optimizing the Control CNN

The purpose of creating a hand optimized CNN is two-fold. First, the process provides a platform for testing different features and understanding how changes impact classification accuracy. This is necessary to understand how to construct the gene pool. Second, it provides the basis for comparison between the two approaches. The hand optimized CNN is the base CNN structure for the genetic algorithm.

A CNN is hand optimized using several on-line resources [5, 6] and class lectures. An iterative approach is used to build the CNN with changes focused first on layer selection then layer configuration back to layer selection then data augmentation features. After each change, 3 – 5 training runs are performed to assess the average impact on classification accuracy. In instances where a change appears to make little difference to the overall classification accuracy, preference is given to the simpler model. The faster training of simpler models help offset the constraints of limited compute resources.

## Using Genetic Algorithms to Tune Parameters

Some mechanism is required to limit the potentially infinite search space from which genes are selected. The output of the hand optimizing experiments provides the constraints to the gene pool. This approach does not encode the CNN features into an alphanumeric or binary string. The Keras models themselves are the chromosomes and genes are retrieved by querying the model attributes.

The genetic algorithm is designed on the model demonstrated in [7]. Numerous mini-runs using simplified parameters for speed allow for quick evaluation of the algorithm and its configured selection and mutation rates. Once the desired evolution behavior is constructed, the complete algorithm is run from start to finish.

A genetic algorithm is of no benefit if it can’t optimize a CNN at least as well as hand optimizing. The classification accuracy of the hand optimized CNN is compared to the performance of the CNN found by the genetic algorithm.

## Visualizing the Resulting Network

Visualization of the genetically optimized CNN provide an additional mechanism to explore the classification efficacy of the CNN. A standard visualization toolkit, keras-vis, developed specifically for Keras provides several types of visualizations. The focus of this paper is the visualization of the output layer. Visualization of the output layer can tell us what the ideal image is for a particular classifier as well as what portions of an input image that classifier focuses on when making its decision.

### Experiments

The project is developed in Google Colaboratory with their Tesla K80 GPU using Python with Keras [8] for neural network support and keras-vis [9] for neural network visualizations. Access to a GPU is necessary to perform network training within a reasonable time. With the GPU, most networks are trained within 5 minutes in comparison to the 15 minutes required without the GPU.

## Hand-tuning the initial CNN

As a starting point, the MNIST CNN Python example supplied with Keras [6] is used. The starting classification accuracy is 99.25%.

The first change is the addition of Data Augmentation. Data augmentation takes random images from the existing dataset and performs transformations based on the input parameters supplied. These additional images are added to the training set. Including additional images with small variations helps the network generalize its learning and perform better with new images it has never seen. The originating data augmentation parameters are taken from [5]. The addition of the data augmentation drops the training classification accuracy by 1% but increases the testing accuracy by 0.2%. Batch sizes of 32 and 128 are tested but their performance do not warrant any changes. At the end of the hand optimization process, various settings for rotation, shear, height and width shifts and zoom, both above and below the original settings are tested. Larger values, resulting in larger variations in the added images, lower accuracy scores. Smaller values, down to 1/3 of the original settings, maintain accuracy rates. The original values are kept as the maximum variation which maintains classification accuracy is the most robust against unknown images.

A third convolutional layer is added along with its companion pooling layer. Accuracy improves therefore it is kept. Adding a fourth convolutional layer lowers classification accuracy and is removed. Adding a second dense layer produces similar classification accuracy therefore it is dropped in favor of the simpler model.

In [5], the author adds a Batch Normalization layer after each convolutional layer. Batch Normalization normalizes the matrix after it has been through a convolution to maintain the scale of each dimension. This modification stabilizes the classification accuracy for both training and testing from run to run. The testing accuracy averages 99.45%. This configuration is rejected because it appears to limit the achievable accuracy. Testing of additional variations in placement of Batch Normalization shows that it is effective when placed immediately after the input layer and the fully connected layer. Classification accuracy averages 99.5%.

A number of layer configurations are tested. Adadelta, Adam and SGD optimizers are examined. Adaldelta and Adam produce similar results but SGD is dropped as its performance is significantly lower. Both Sigmoid and Softmax are tested for the final layer activation. Sigmoid is dropped for performance reasons. Different pooling sizes, strides, kernel sizes, dropout rates, convolutional kernels and epochs are tested. Most of the values tested produce results between 99.3% and 99.55%.

The highest classification accuracy seen during the hand optimization is 99.59%. The final CNN, used for the genetic algorithm, is diagramed in Fig. 4.

## The Gene Pool

The designing of the initial CNN provided considerable insight into the feature selections for the gene pool. The testing of different features during hand optimization highlighted the range of values likely to produce good results. Any value used during the design of the CNN structure that did not significantly degrade accuracy is included in the gene pool. The first few tests runs to confirm that random selection of values from the gene pool worked identified an unfortunate limitation of Google Colaboratory. When multiple convolutional layers are assigned convolutional kernels in excess of 256 Colaboratory terminates the program with a Resource Exhaustion: OOM error. To prevent the program from potentially terminating after hours of computation, the first two convolutional layers are limited to 128 convolutional kernels and the last layer to 512.

The gene pool is listed in Table I. A total of 933,120 different networks are producible using the hand optimized CNN structure with the selected gene pool.

TABLE I

Gene Pool used for generation of network population

|  |  |
| --- | --- |
| Genes | Values for selection |
| Conv2D Kernels | 32, 64, 128, 256, 512 |
| Conv2D Kernel Size | 2, 3, 4 |
| Strides | 1, 2 |
| Dropout Rates | 0.2, 0.3, 0.4, 0.5 |
| Hidden Units | 128, 256, 512 |
| Optimizer | Adadelta, Adam |
| Epochs | 6, 12, 18 |

Each individual is made up of 12 genes whose values are randomly picked from the gene pool for the starting population or from the parents in successive generations. With the exception of the Epochs gene, the genes are stored in the Keras neural network model definition for that individual. The Epochs gene is stored in a list whose order matches the order of the population list. At the point of mutation or breeding, genes are retrieved from the model and placed in a list in a predefined order like a chromosome. This chromosome is used for programmatic manipulation, and then discarded when no longer needed. Table II shows the order of the genes in the chromosome.

TABLE II

Individual Chromosome Layout

|  |  |
| --- | --- |
| Genes | Gene Pool Values |
| Conv2\_1 Kernels | Conv2D Kernels [0:3] |
| Conv2\_1 Kernel Size | Conv2D Kernel Size |
| Conv2\_2 Kernels | Conv2D Kernels [0:3] |
| Conv2\_2 Kernel Size | Conv2D Kernel Size |
| Conv2\_3 Kernels | Conv2D Kernels |
| Conv2\_3 Kernel Size | Conv2D Kernel Size |
| Conv2\_1 Strides | Strides |
| Dropout\_1 Rate | Dropout Rates |
| Dropout\_2 Rate | Dropout Rates |
| Dense\_1 Hidden Units | Hidden Units |
| Optimizer | Optimizers |
| Epochs | Epochs |

## The Genetic Algorithm

Evolution starts with a randomly generated population and a method of evaluating the individual’s fitness for inclusion in the next generation. The starting population of 20 networks suggested in [7] is used. Individual fitness evaluation is based entirely on the classification accuracy computed for the testing dataset.

The best 25%, ranked by the fitness function, are automatically selected to populate the next generation. In addition, each remaining network, regardless of fitness, has a 25% chance of joining the next generation. On average, 4 of the remaining individuals are included in the next generation. Inclusion of the some of the worst performers allows for lucky combinations of the best and worst preventing convergence on a local maximum. Mutations of genes are another method of moving a population from a path of convergence to a local maximum. Each of the selected individuals has a 15% chance of a single gene mutation. If an individual is chosen for mutation, a single gene is randomly selected for mutation and the new value is randomly picked from the gene pool. That new value could be the same as the current value and no mutation actually occurs. Depending on the gene, the chance of no mutation ranges from 20-50%. On average, 1 mutation occurs every generation.

With the possible parents selected and mutations, if any, applied, children are now generated to replace the rejected networks. Two parents are randomly picked from the retained networks. Parents are constrained to uniqueness. As a result, if the algorithm randomly selects the same network for both parents, the selection is discarded and new parents are picked. Each child gene is randomly selected from a parent until the entire chromosome is filled in. The child network is then trained.

The genetic algorithm is configured to iterate for a number of generations. Only 5 generations of evolution are performed. This number is determined solely by Google Colaboratory. Colaboratory limits the running time of a notebook to 12 hours automatically terminating the notebook when the time limit is reached. Total running time for 5 generations is estimated at approximately 8 to 9 hours. Each network takes, on average, 5 to 6 minutes to train. A starting population of 20 networks should take almost 2 hours of computational time. Each generation is estimated to include 2 mutations and the creation of 12 children for a total of 70 minutes of computation. Rather than risk losing a day of computation if the program overruns the estimates, a conservative number is selected.

The 20 network models contained in the 5th generation are saved to Google Drive at the end of processing. It was hoped these models could be reloaded as the starting population for another set of 5 generations. Loading the saved models produces an error indicating that the optimizer object can’t be loaded and a new optimizer object is created. At first glance, this doesn’t appear to be an issue as the default optimizer is used. Unfortunately, Google Colaboratory terminates the notebook with no errors in the first generation while creating child networks. After several attempts, extending the completed 5 generations another 5 generations is abandoned.

## Visualizing the Resulting Network

During the hand and genetic optimization, various models are saved to disk. Any of these models can be reloaded for further processing including visualization. The very first step in visualization is the removal of the softmax activation on the output layer. Kera-vis provides a utility to perform this function which doesn’t require retraining the entire model.

Of interest is the visualization of the ideal images for each of the classifiers. Despite using the keras-vis MNIST example [6] for loading MNIST data into the program structures, the default input range of (0,255) and their suggested range of (0, 1) does not provide good visualizations for the models developed for this project. An input range of (0, 1) produces a black square for every classification. Various input ranges are tested and the range (-.5, .75) is selected. A comparison of how input range affects the visualization of the classifier is shown in Figure 1.

A second visualization, the saliency map, shows the parts of the input image that activate a filter or classifier. When used for visualizing the output layer, it shows the pixels that contribute to the classification of the image. A saliency map is a useful tool to determine if the neural network is focusing on the right part of the image when performing its classification. In [10] is an example of using saliency to highlight a classification issue.

Keras-vis produces saliency maps using a ‘Jet’ color map. For the colorblind this particular coloration can be difficult to see. In general, it is difficult to discern patterns in visualizations with this color map. Several color maps were tested before selecting ‘Inferno’. Figure 2 shows saliency maps and ideal images in both ‘Jet’ (upper) and ‘Inferno’ (lower). ‘Inferno’ is applied to all visualization outputs for consistency

|  |  |
| --- | --- |
|  |  |

Figure 1: Visualization of the 0 classifier using different input range modifiers. The left image is produced by input range of (-.5, .75). The right image is produced by the default value.

|  |
| --- |
|  |
|  |

Figure 2: Comparison of default saliency color map and 'Inferno' color map. The left image is a saliency map and the right image the ideal image for the digit 3 classifier.

The saliency maps produced by keras-vis are pixelated with many different values side-by-side. A smoother output would direct the attention to the areas of interest. Using the example provided by [10], a Gaussian blur is added to the saliency map. Figure 3 provides a comparison between the standard output and the output with two different Gaussian filters configurations applied. The Gaussian filter removes a lot of the noise highlighting the strongest parts of the saliency map. For this application, a Gaussian filter with a sigma of 1 is used. Due to the small size of the images, a larger sigma obscures some of the other smaller details.

|  |
| --- |
|  |

Figure 3: Comparison of the default Saliency Map and Saliency Maps with Gaussian filters applied. Left is the default output, middle with Gaussian filter of sigma =1 and right with Gaussian filter of sigma=2 applied.

### Results

A best effort is made to improve the starting CNN architecture with approximately 30 different networks tested. The final CNN architecture selected has a classification accuracy of 99.59%; a significant improvement over the starting accuracy of 99.25%.

A discovered genetic algorithm should be able to perform at least as well as a manually hand optimized CNN. Ideally, with a sound evolution strategy, a better-performing CNN should be discovered. After 5 generations of evolution, 78 networks out of the possible 933,120 combinations are created and evaluated. Table III shows the networks kept, mutated and created during the evolutionary process.

In several of the mini test runs, the mutation is applied to the fittest individual. This set back the evolution process considerably reducing the classification accuracy as a whole in the next generation. A better strategy would have been to exclude the fittest individual from the mutation process. This does not impact the final results as none of the fittest individuals in any generation are mutated in the final run.

Table III

Evolution Activity by Generation

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Generation | Retained | Random Selects | Mutations | Children |
| Initial Pop | 20 | - | - | - |
| 1 | 5 | 3 | 1 | 12 |
| 2 | 5 | 4 | 0 | 11 |
| 3 | 5 | 5 | 0 | 10 |
| 4 | 5 | 4 | 1 | 11 |
| 5 | 5 | 5 | 2 | 10 |

After each consecutive generation the classification accuracy should improve. As poor performers are set aside and the fittest individuals kept to pass on their genes to new networks, the classification accuracy of the population as a whole should improve. Table IV shows the genetic algorithm improving over the generations but the results are mixed. The maximum classification accuracy achievable increases over the evolution process but the average accuracy of the generation’s population starts to flatten. Although the planned evolution was for 10 generations, it appears there is diminishing returns to continued investment in the computational time required for the evolution process. The final model used for the visualizations achieved a classification accuracy of 99.67%. The hand optimized and genetically optimized CNNs are compared in Fig. 4.

Table IV

Classification Accuracy by Generation

|  |  |  |  |
| --- | --- | --- | --- |
| Generation | Maximum Accuracy | Minimum  Accuracy | Average Accuracy |
| Initial Pop | 99.63 | 98.6 | 99.26 |
| End of 1 | 99.63 | 99.32 | 99.51 |
| End of 2 | 99.63 | 99.34 | 99.52 |
| End of 3 | 99.64 | 99.43 | 99.55 |
| End of 4 | 99.64 | 99.42 | 99.55 |
| End of 5 | 99.67 | 99.2 | 99.53 |

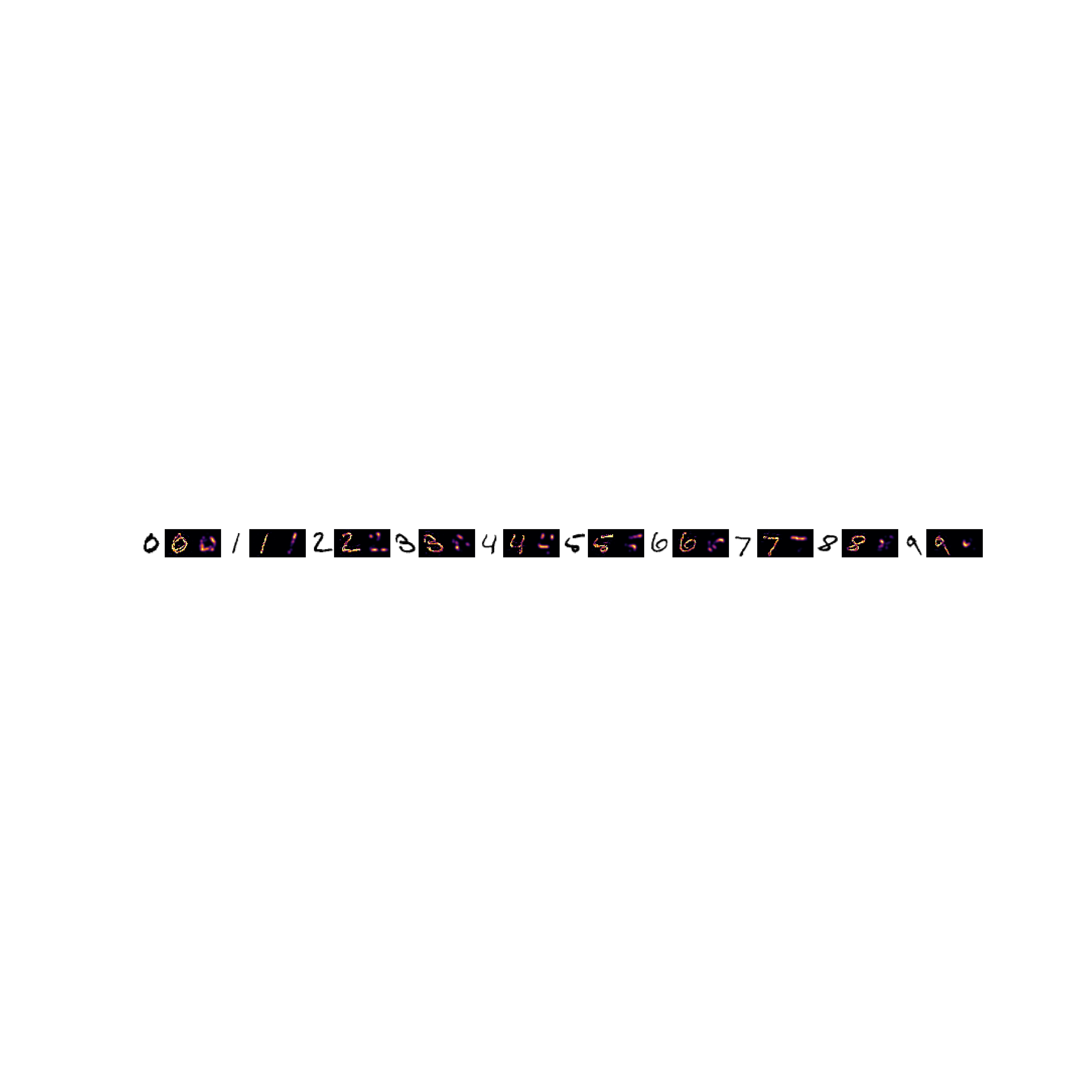
|  |  |  |
| --- | --- | --- |
| Hand-Tuned  Configuration | Architecture | Evolved  Configuration |
|  |  |  |
| 32 (4,4) Kernels | Conv2D | 32 (2,2) Kernels |
|  |  |  |
|  | Batch Normalization |  |
|  |  |  |
| 64 (3,3) Kernels | Conv2D | 32 (4,4) Kernels |
|  |  |  |
|  | Pooling (2 x 2) |  |
|  |  |  |
| 128 (2,2) Kernels | Conv2D | 512 (3,3) Kernels |
|  |  |  |
|  | Pooling (2 x 2) |  |
|  |  |  |
| 25% | Dropout | 50% |
|  |  |  |
|  | Flatten |  |
|  |  |  |
| 256 | Dense | 512 |
|  |  |  |
|  | Batch Normalization |  |
|  |  |  |
| 50% | Dropout | 40% |
|  |  |  |
|  | Dense  Nh-10, softmax |  |
|  |  |  |
| Adam | Optimizer | Adadelta |
|  |  |  |
| 12 | Epochs | 12 |

Figure 4: Comparison of Hand and Genetically Optimized CNN

Visualizations of the output layer for well-trained networks should show ideal images for classifiers that have strong resemblance to the actual images being classified. The ideal classifier images do not replicate any handwritten image in the dataset but rather displays the characteristics of the ideal image. When the variety in shape and size of the written digits are considered, these images start to make sense. For example, the digit 0 may be written tall and thin or short and fat. When looking at the elongated shape in Fig. 5 with appendages sprouting between the top and bottom curves, one can imagine that the 0 can be completed down along any of the appendages creating thinner or wider circles.

Figure 5 compares the clarity of the hand optimized images against the genetically optimized CNN. The images from the genetically optimized CNN have stronger resemblances to the actual digits showing the better training achieved by the genetic algorithm.

Visualizations of this type can be performed at any layer indicating the shapes that trigger a particular convolutional kernel. In the earlier layers of the genetically optimized CNN, many of the visualizations are in primarily in black and white rather than the expected coloration. The closer the layer being visualized is to the output layer, the more the visualizations are in color. The black and white visualizations may indicate that the kernel is not strongly attracted to a particular pattern. Fig. 7 shows the increasing complexity of the patterns that trigger the convolutional kernels as the CNN moves from the first to the last convolutional layer.



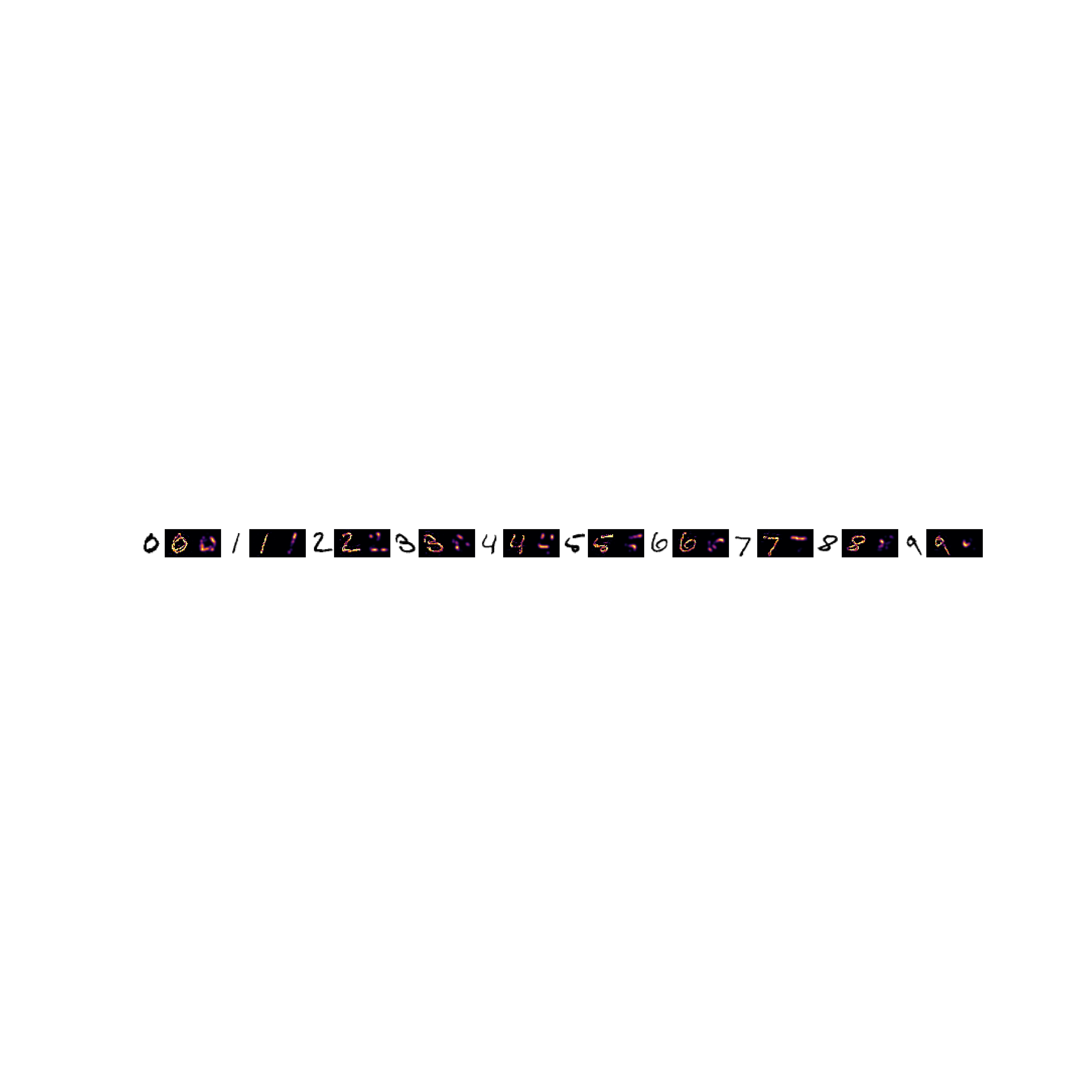
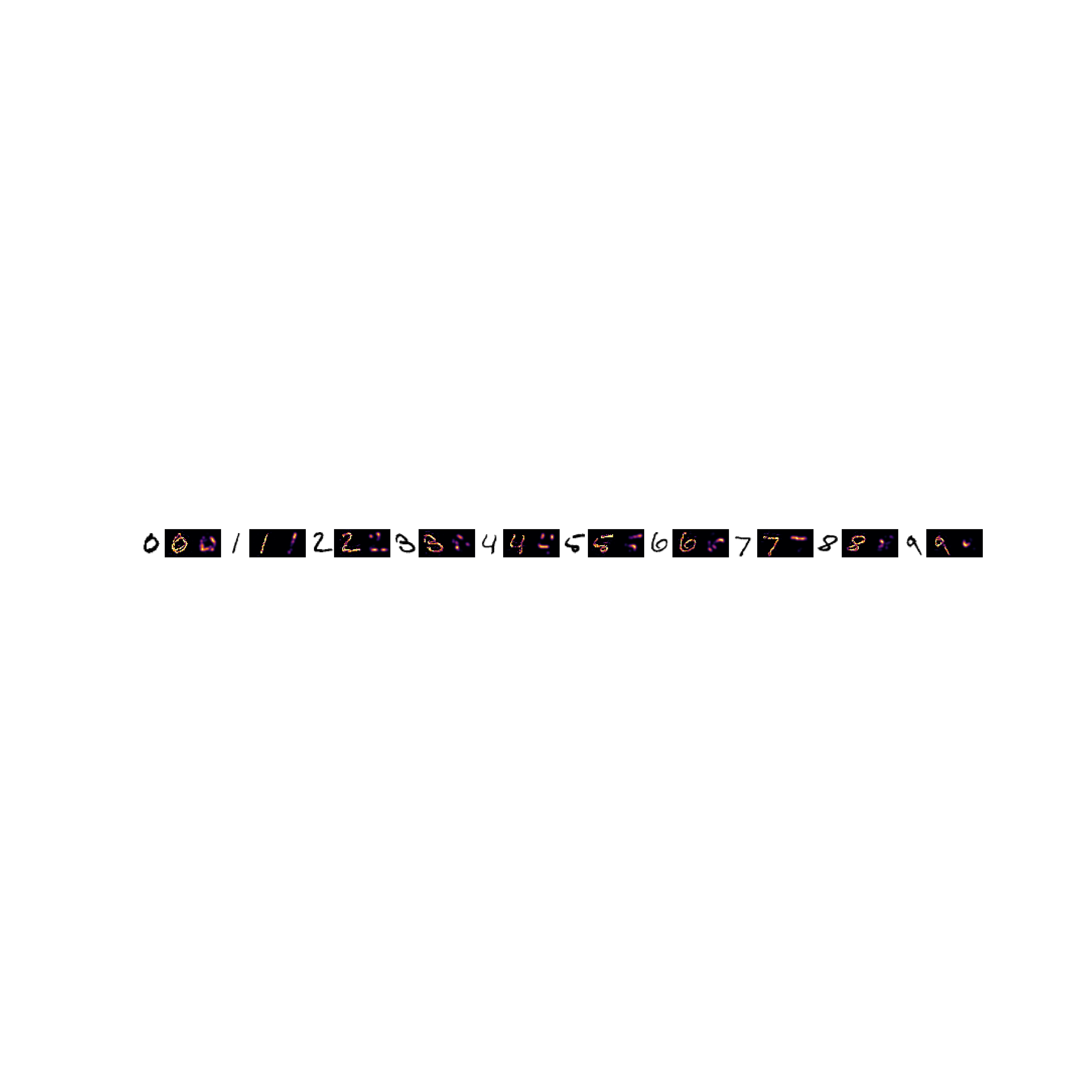
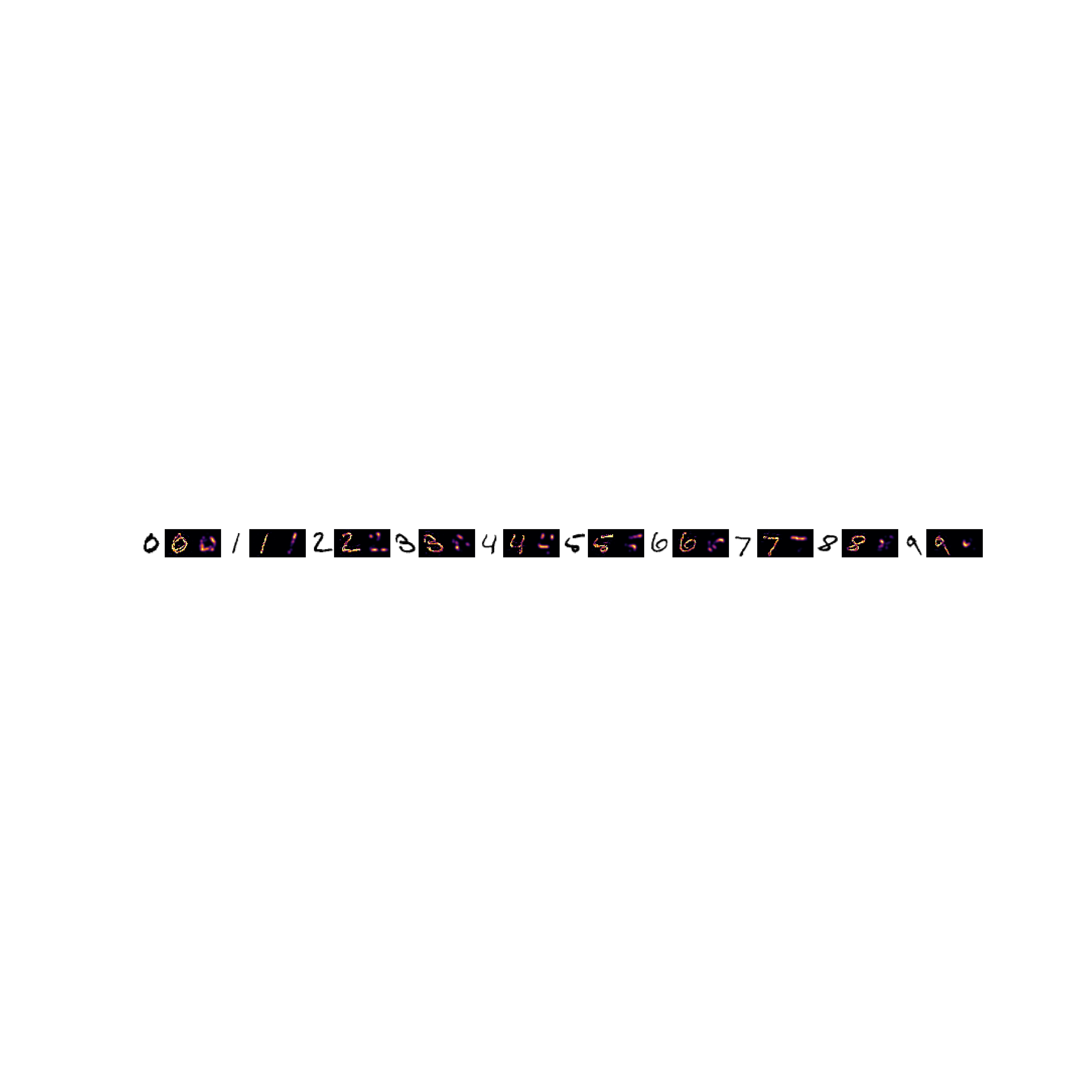
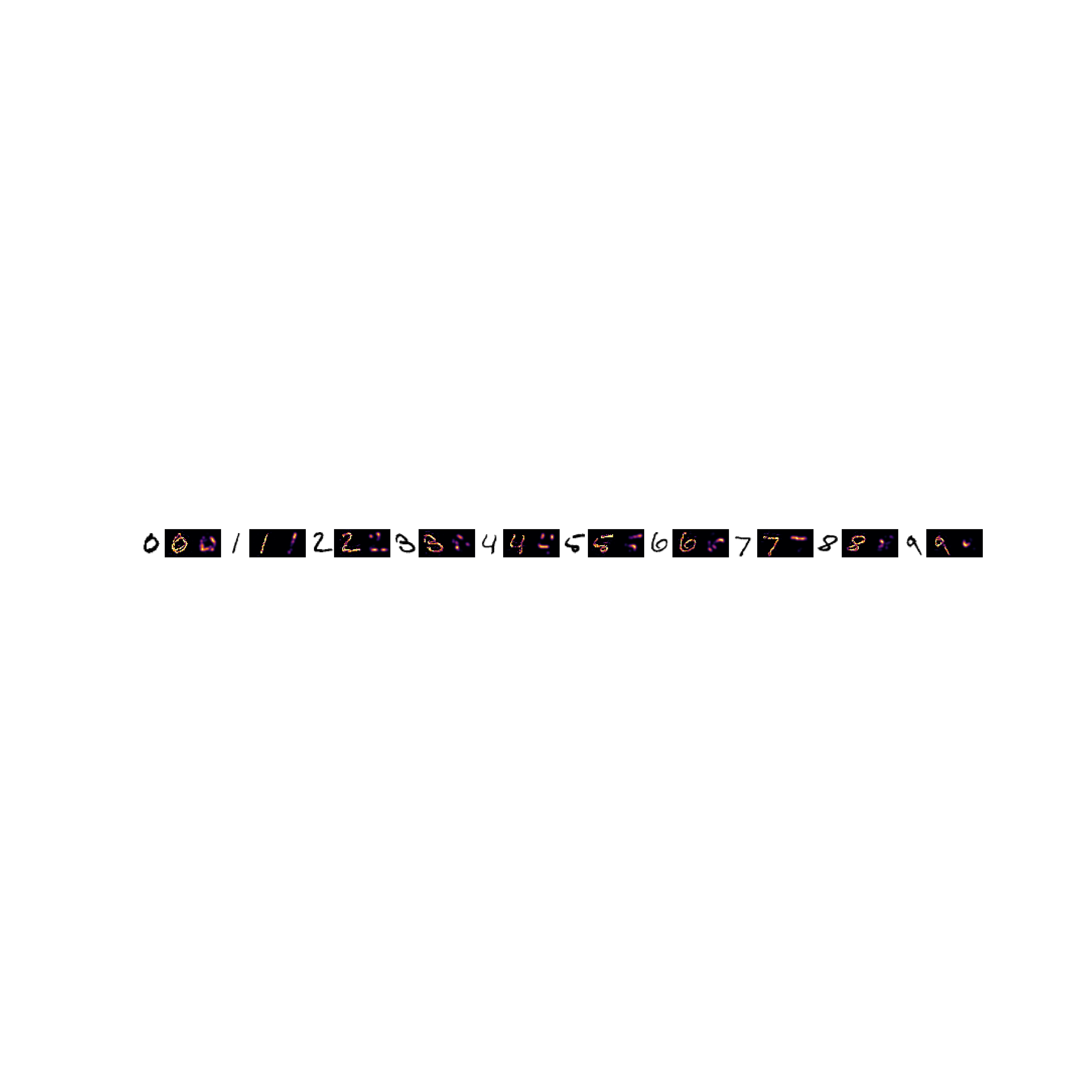


Figure 8: Saliency Maps of selected digits from MNIST dataset. Each digit has a trio of images. From left to right for each digit: input image, default Saliency Map, Saliency Map with Gaussian blur applied.



Figure 5: Visualizations of images best suited to triggering the classifier for that handwritten digit. From left to right are digits 0 to 9 Top row is produced from the hand optimized CNN. Bottom row is produced from the genetically optimized CNN.

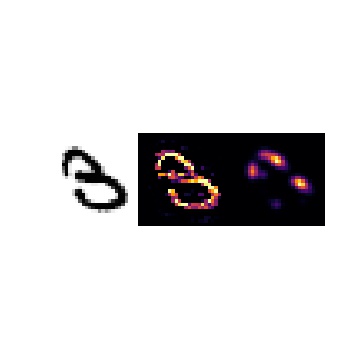


Figure 6: Pixel focus does not follow classification. Top row shows Saliency Map for digit 3 classifier. Bottom row: Saliency map using digit 8 classifier

|  |
| --- |
| C:\Users\andria\AppData\Local\Microsoft\Windows\INetCache\Content.Word\VisualizeFilterActivation-conv2_1-inferno-5+75-relu.jpg |
| C:\Users\andria\AppData\Local\Microsoft\Windows\INetCache\Content.Word\VisualizeFilterActivation-conv2_2-inferno-5+75 (1).jpg |
| C:\Users\andria\AppData\Local\Microsoft\Windows\INetCache\Content.Word\VisualizeFilterActivation-conv2_3-inferno-75+75-relu.jpg |

Figure 7: Visualizations of the patterns that will trigger the convolutional kernel (filter). Top to bottom: first convolutional layer to third convolutional layer

Also of interest are saliency maps that indicate which pixels in an image activate the classifier. In general, the classifier focuses on appropriate image pixels for the classification. Most of the digits in Fig. 8 are clearly outlined in the saliency map. Only digits 7 and 9 show weak activation although they still follow the contours of the image.

Some anomalies are found with the saliency maps not matching the image classification. For example, image #18 in the test dataset, a 3, has no pixels which activate the classifier, yet, the image is correctly classified. This same image activates the digit 8 classifier. It is not clear why the image is classified as a 3 and not an 8. See Fig. 6

### Conclusion

A genetic algorithm can design better convolutional neural networks. The tests conducted on the MNIST handwritten digits show that a genetically optimized CNN can improve classification accuracy when compared to a hand optimized model. The genetically optimized model achieved a classification accuracy of 99.67% in comparison to the 99.59% achieved by the hand optimized model. The visualization comparison of the output layers of both networks confirm that the genetically optimized model is better trained as its activation patterns more closely resemble the original input images.

### Future Work

The genetic algorithm in this paper is limited to optimizing the configuration of the existing layers in the hand optimized CNN using a very simple algorithm. Obviously, the structure of the CNN impacts the capability of the network to perform image classification. A more comprehensive study of genetically optimizing all of the CNN features is worth exploring to determine if genetic algorithms are successful on a more complex scale. Multiple strategies exist to perform the mechanics of evolution from selection of parents to recombination of genes. Different strategies from [11] could be applied to examine what impact they have on the final solution.

Data augmentation parameters also influence the image classification of the CNN. Adding these parameters to the gene pool enlarges the search space exponentially. A more interesting use would be to test the robustness of the genetically optimized CNN against the varying data augmentation strategies. The fitness evaluation would then be based on an average accuracy over several different data generators with CNNs more robust to varying training sets ranking higher.

# References

|  |  |
| --- | --- |
| [1] | S. A. Harp and T. Samad, "Optimizing Neural Networks with Genetic Algorithms," in *Neural Network Computing for the Electric Power Industry: Proceedings of the 1992 INNS Summer Workshop*, Lawrence Erlbaum Associates, Inc. , 1993, pp. 41-44. |
| [2] | D. Whitley, T. Starkweather and C. Bogart, "Genetic algorithms and neural networks :Optimizing connections and connectivity," *Parallel Computing,* pp. 347-361, August 1990. |
| [3] | A. J. Reiling, "CONVOLUTIONAL NEURAL NETWORK OPTIMIZATION USING GENETIC ALGORITHMS," University of Dayton, Dayton, 2017. |
| [4] | L. Xie and A. Yuille, "Genetic CNN," arXiv:1703.01513 [cs.CV], 03 2017. [Online]. Available: https://arxiv.org/pdf/1703.01513.pdf. [Accessed 29 4 2018]. |
| [5] | Y. Katariya, "Applying Convolutional Neural Network on the MNIST dataset," 15 4 2017. [Online]. Available: https://yashk2810.github.io/Applying-Convolutional-Neural-Network-on-the-MNIST-dataset/. [Accessed 29 4 2018]. |
| [6] | F. a. o. Chollet, "Keras example for MNIST," [Online]. Available: https://github.com/keras-team/keras/blob/master/examples/mnist\_cnn.py. [Accessed 28 4 2018]. |
| [7] | M. Harvey, "Let’s evolve a neural network with a genetic algorithm—code included," Coastline Automation, 6 4 2017. [Online]. Available: https://blog.coast.ai/lets-evolve-a-neural-network-with-a-genetic-algorithm-code-included-8809bece164. [Accessed 28 4 2018]. |
| [8] | F. a. o. Chollet, "Keras," 2015. [Online]. Available: https://keras.io. [Accessed 2018 28 4]. |
| [9] | R. a. c. Kotikalapudi, "keras-vis," 2017. [Online]. Available: https://github.com/raghakot/keras-vis. [Accessed 28 4 2018]. |
| [10] | B. Sliz, "Where the Machines are Looking.," 12 12 2017. [Online]. Available: http://bradsliz.com/2017-12-21-saliency-maps/. [Accessed 1 5 2018]. |
| [11] | B. J. Lynch, "Optimizing with Genetic Algorithms," 23 2 2006. [Online]. Available: https://www.msi.umn.edu/sites/default/files/OptimizingWithGA.pdf. [Accessed 28 4 2018]. |