Parameter efficient dendritic-tree neurons outperform perceptrons

Ziwen Han [*1] **Evgeniya Gorobets** [*1] **Pan Chen** [1]

Abstract

Biological neurons are more powerful than artificial perceptrons, in part due to complex dendritic input computations. Inspired to empower the perceptron with biologically inspired features, we explore the effect of adding and tuning input branching factors along with input dropout. This allows for parameter efficient non-linear input architectures to be discovered and benchmarked. Furthermore, we present a PyTorch module to replace multi-layer perceptron layers in existing architectures. Our initial experiments on MNIST classification demonstrate the accuracy and generalization improvement of dendritic neurons compared to existing perceptron architectures.

1. Introduction

Many artificial neural networks (ANNs) include variants of the perceptron [10], which takes a linear combination of input signals and applies a nonlinear activation function to produce an output signal. More recent neuroscience research has revealed that the dendrites of a biological neuron perform multiple complex nonlinear computations on their input signals [6], as opposed to a linear function. Furthermore, neuroscientists have advocated for incorporating dendritic features to improve existing ANN performance [1]. Empirically, Jones and Kording have demonstrated that a single dendritic neuron model with input repetition (k-trees) can reach accuracy similar to multi-layer perceptrons (MLPs) of similar parameter size on binary image classification tasks. [4]. Accordingly, we hypothesize multiple artificial dendritic neurons working in conjunction could be more powerful than their MLP counterpart beyond binary tasks. Current dendritic models are either limited by structural rigidity or fail to incorporate the tree-like structure of biological dendrites, which may not capture the full breadth of dendritic computation.

[*]Equal contribution [1]Department of Computer Science, University of Toronto, Toronto, Canada. Correspondence to: Ziwen Han <ziwen.han@mail.utoronto.ca>.

DyNN workshop at the 39^{th} International Conference on Machine Learning, Baltimore, Maryland, USA, 2022.

2. Related Works

2.1. Neuron Models

Multiple works have proposed ANNs based on neuron models that simulate dendritic input. These alternatives include dendrite morphological neural networks (DMNNs) [9], dendritic neural networks (DENNs) [14], the single dendritic neuron model (DNM) [12], and most recently the model by Jones and Kording based on a balanced tree structure [4]. Our model expands on the work of Jones and Kording by: (1) exploring the effect of generalizing the dendritic tree structure to allow tunable branching, dropout, and activations without k-tree redundancy; (2) using layers of dendritic neurons for non-binary classification tasks and evaluate overfitting; (3) attaching the layer of neurons to a CNN to evaluate performance as a perceptron replacement.

2.2. Multi-Neurons

Other studies have connected dendritic neurons in MLP-like architectures [9, 14], including a hybrid DNM-CNN adaptation of the DNM model [13]. Each of these multi-neuron architectures uses a fundamentally different neuron model. DNM neurons connect each dendritic branch to each input and rely on logical operations, while DMNNs utilize a different underlying mathematical structure. By contrast, our model enforces sparse, localized connections between dendrites and inputs in a tree structure, which more closely models the spatially-limited connections between biological neurons. The DENN model enforces dendrite-input sparsity, but uses one-layer dendrite trees [14]. Our dendritic trees are deeper, to more closely replicate the anatomy and complexity of biological neural networks.

3. Methods

3.1. Model Architectures

Using PyTorch (1.10.0+cu111) [7], we implemented a dendritic tree neuron based on the Jones-Kording single neuron model [4] (Figure 2 in the Appendix). Our implementation generalizes the original architecture by allowing users to specify the branching factor and number of neurons in the `DendriticLayer`. Each `DendriticLayer` has a constant branching factor, but multiple instances of the module

can be stacked together to achieve different branching per layer. To vectorize the `DendriticLayer`, we treat the inputs to all the neurons as a single input tensor, and we compute the next layer of all dendrite trees simultaneously. The tree structure is enforced by constructing a mask for the weight matrix at each layer (Figure 1). Table 1 describes the full architecture of the `DendriticLayer` module.

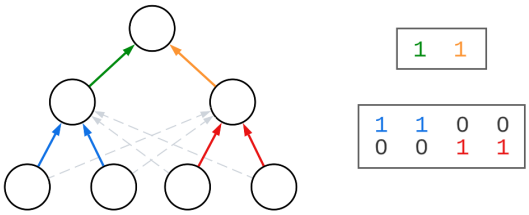

Figure 1. An example of the mask used to enforce the tree structure in our implementation. The tree structure on the left represents the dendritic neuron ($d = 2, b = 2$), rectangles on the right represent the mask matrices for each tree layer. The mask and weight matrices are multiplied element-wise. Colored arrows indicate preserved weights/connections, masked connections are represented by dashed gray arrows corresponding to zeroes.

We used Jones and Kording's modified density gain Kaiming He initialization scheme to account for tree sparsity and stabilize training [4, 3]. Each weight in the weight matrices is initially sampled: $\mathbf{W}_{jk}^{(i)} \overset{iid}{\sim} \mathcal{N}(0, \frac{2}{I \cdot b^{-i+1} \cdot \text{density}})$, where density $= \frac{I \cdot b^{-i+1}}{I \cdot b^{-i+1} \cdot I \cdot b^{-i}}$, and $I \cdot b^{-i+1}, I \cdot b^{-i}$ are the number of input and output units in the i^{th} layer of the dendritic tree.

Using the `DendriticLayer` as our base, we built several multi-layer neuron (MLN) architectures on top of it. (1) An `MLNBinaryClassifier`, which is a single neuron that predicts a binary output activated using a logistic sigmoid function. (2) An `MLNClassifier`, which repeats the same set of inputs to a specified number of neurons (one for each class) and returns the output of each neuron activated through a softmax function. (3) A `ConvMLN`, which runs the inputs through a CNN (described in Section 9.3 in the Appendix), then flattens the output of the convolutional layers and feeds it to a single layer of neurons for classification. All models are equipped with a tunable input dropout layer to be robust against overfitting [11].

3.2. Computational Tasks and Controls

We tested our single neuron models on binary classification tasks, using a subset of MNIST that consisted only of images labeled as "4" or "9" (referred to as 4-9 MNIST) [2]. We used the full MNIST dataset to test the classification capabilities of our multi-neuron architectures [2].

For controls, we constructed multi-layer perceptrons (MLPs) that performed the same tasks as each of our dendritic models. The architecture of the MLPs is described in Table 2. The number of hidden units (h) was modified in order to match the number of parameters in the dendritic models. The number of parameters in each model is listed in Tables 9, 10, and 11 in the Appendix. Each model was initialized using the Kaiming He method, but without density gain since MLPs are not sparse [3].

3.3. Data Preprocessing and Results Analysis

The standard MNIST dataset images are 28×28. Nearest neighbour upsampling was used to scale inputs to 32×32, to better fit the dendritic branching factors. For non-CNN architectures, input flattening was applied to create a 1024-dimension tensor.

To aggregate results from multiple trials, the epoch with the lowest validation loss was taken from each trial as the best performance to evaluate at. Post-experiment statistical analysis was conducted using R [8].

3.4. Model Training

All models were trained on the Google Colab environment with CUDA for 100 epochs, using a batch size of 128. Every model was re-initialized and trained 10 times. The learning rates used for dendritic MLN and MLP models were 0.05 and 0.001, respectively. All models used either binary or categorical cross entropy loss. All models were trained using the Adam Optimizer [5].

4. Results

4.1. Single Neuron Binary Classification

We modified the Jones-Kording single neuron model to investigate the effects of branching factors (b) and dropout probabilities (p) (Table 3). The control MLPs with similar numbers of parameters are in Table 4. Dropout models were tested with p = 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, but only the best-performing set of models are reported in each experiment.

4.2. Multi-Neuron Classification

To test non-binary classification, we integrated multiple neurons in a layer (`MLNClassifier`). Each neuron connected to the same set of inputs and was expected to predict a single MNIST digit. The experimental setup identical to the binary case, as described in Sections 3.4, 4.1. The average performance of each multi-neuron model and each control MLP is reported in Tables 5 and 6.

$\mathbf{y} = f^{(d)} \circ f^{(d-1)} \circ \cdots \circ f^{(1)}(\mathbf{x})$	$\mathbf{x} \in R^I, \mathbf{y} \in R^O -$ the input and output tensors
$f^{(i)}(\mathbf{z}) = \text{LeakyReLU}((\mathbf{W}^{(i)} * \mathbf{M}^{(i)})\mathbf{z} + \mathbf{b})$	$f^{(i)} : R^{I \cdot b^{-i+1}} \to R^{I \cdot b^{-i}} -$ the function for the i^{th} layer of the dendrite tree
	$\mathbf{W}^{(i)}, \mathbf{M}^{(i)} \in R^{I \cdot b^{-i} \times I \cdot b^{-i+1}} -$ the weight and mask matrices
	$\mathbf{b}^{(i)} \in R^{I \cdot b^{-i}} -$ the bias tensor
$I = Ob^d$	$b -$ the branching factor; $d -$ the depth of the dendrite tree (# edges)

Table 1. Equations describing the `DendriticLayer` architecture. The $*$ denotes element-wise matrix multiplication. The LeakyReLU activations used a negative slope value of 0.1.

	Binary MLP	**Multiclass MLP**
Equation	$y = \sigma\big(\mathbf{W}^{(2)}\big(\text{ReLU}(\mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)})\big) + b^{(2)}\big)$	$\mathbf{y} = \text{softmax}\big(\mathbf{W}^{(2)}\big(\text{ReLU}(\mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)})\big) + \mathbf{b}^{(2)}\big)$
First Layer	$\mathbf{W}^{(1)} \in R^{h \times bn}, \mathbf{b}^{(1)} \in R^h$	$\mathbf{W}^{(1)} \in R^{h \times bn}, \mathbf{b}^{(1)} \in R^h$
Second Layer	$\mathbf{W}^{(2)} \in R^{1 \times h}, b^{(2)} \in R$	$\mathbf{W}^{(2)} \in R^{10 \times h}, \mathbf{b}^{(2)} \in R^{10}$
Output	$y \in [0, 1]$	$\mathbf{y} \in [0, 1]^{10}$

Table 2. Equations describing control MLP architectures

Single Neuron	b	p	**Train Acc.**	**Val. Acc.**
1	2	0	0.91 ± 0.06	0.85 ± 0.09
2	4	0	0.94 ± 0.06	0.89 ± 0.06
3	32	0	0.99 ± 0.02	0.91 ± 0.02
4	2	0.4	0.85 ± 0.07	0.89 ± 0.07
5	4	0.5	0.89 ± 0.03	0.92 ± 0.02
6	32	0.3	0.96 ± 0.02	0.92 ± 0.02

Table 3. Mean performance $\pm$ standard deviation of a single dendritic neuron (`MLNBinaryClassifier`) on 4-9 MNIST. Models differ in their branching factor (b) and dropout probability (p).

Binary MLP	h	p	**Train Acc.**	**Val. Acc.**
1	3	0	0.95 ± 0.15	0.88 ± 0.13
2	2	0	0.86 ± 0.22	0.79 ± 0.19
3	3	0.6	0.95 ± 0.02	0.92 ± 0.01
4	2	0.4	0.97 ± 0.01	0.91 ± 0.01

Table 4. Mean performance $\pm$ standard deviation of binary MLPs on 4-9 MNIST. $h =$ number of hidden units; $p =$ dropout probability. MLPs with $h = 3$ were controls for neurons with $b = 2$; MLPs with $h = 2$ were controls for neurons with $b = 4, 32$ (smallest possible two-layer MLP).

MLN	b	p	**Train Acc.**	**Val. Acc.**
1	2	0	0.75 ± 0.07	0.59 ± 0.04
2	4	0	0.90 ± 0.06	0.67 ± 0.05
3	32	0	0.94 ± 0.05	0.77 ± 0.02
4	2	0.2	0.77 ± 0.06	0.62 ± 0.04
5	4	0.6	0.79 ± 0.07	0.75 ± 0.06
6	32	0.4	0.97 ± 0.02	0.82 ± 0.02

Table 5. Mean performance $\pm$ standard deviation of a ten-neuron single-layer model on MNIST. The models differ in their branching factor (b) and dropout probability (p).

Multiclass MLP	h	p	**Train Acc.**	**Val. Acc.**
1	30	0	0.93 ± 0.07	0.63 ± 0.07
2	14	0	0.87 ± 0.08	0.58 ± 0.07
3	11	0	0.87 ± 0.11	0.58 ± 0.04
4	30	0.5	0.92 ± 0.04	0.66 ± 0.04
5	14	0.1	0.89 ± 0.07	0.61 ± 0.06
6	11	0.3	0.86 ± 0.08	0.59 ± 0.07

Table 6. Mean performance $\pm$ standard deviation of MLPs on MNIST. $h =$ number of hidden units; $p =$ dropout probability. MLPs with $h = 30, 14, 11$ were controls for MLNs with $b = 2, 4, 32$, respectively.

CNN-MLN	b	p	**Train Acc.**	**Val. Acc.**
1	2	0	0.82 ± 0.07	0.61 ± 0.08
2	4	0	0.96 ± 0.03	0.74 ± 0.04
3	16	0	0.99 ± 0.02	0.78 ± 0.03
4	2	0.1	0.77 ± 0.07	0.64 ± 0.07
5	4	0.2	0.93 ± 0.04	0.81 ± 0.04
6	16	0.5	0.95 ± 0.02	0.83 ± 0.03

Table 7. Mean performance $\pm$ standard deviation of CNN-MLN models on MNIST classification. The models differ in their branching factor (b) and dropout probability (p).

4.3. Dendritic Models with CNNs

To explore the potential as a tunable, modular MLP replacement in existing architectures, we connected 10 neurons to a simple CNN architecture for MNIST classification (`ConvMLN`) as a replacement to the usual perceptrons. This experimental setup is described in Sections 3.4, 4.1. The performance of each CNN-MLN and control CNN-MLP is listed in Tables 7and 8.

CNN-MLP	h	p	Train Acc.	Val. Acc.
1	29	0	0.97 ± 0.04	0.7 ± 0.05
2	16	0	0.99 ± 0.03	0.7 ± 0.06
3	11	0	0.94 ± 0.06	0.64 ± 0.02
4	29	0.1	0.99 ± 0.02	0.74 ± 0.06
5	16	0.4	0.98 ± 0.02	0.74 ± 0.06
6	11	0.1	0.96 ± 0.06	0.67 ± 0.07

Table 8. Mean performance $\pm$ standard deviation of CNN-MLPs on MNIST. h = # of hidden units; p = dropout probability. CNN-MLPs with $h = 29, 16, 11$ were controls for CNN-MLNs with $b = 2, 4, 16$, respectively.

4.4. Results Summary

Both increasing branching factor and introducing dropout significantly improved performance. The best performing dendritic models were two layers deep, with $b = \sqrt{i}$, where i is number of inputs to each neuron and $p \geq 0.1$. We hypothesize deeper trees led to vanishing gradients; shallow dendritic trees trained and performed better despite having fewer parameters. In all experiments, dendritic neuron models with $b = 2$ perform worse than their control MLPs, but neurons with $b > 2$ surpass their control MLPs in terms of validation performance and robustness to overfitting, both with and without dropout. This shows dendritic trees can improve performance while maintaining parameter efficiency relative to MLP counterparts.

5. Discussion

We created an encapsulated generalized dendritic-tree neuron inspired by Jones and Kording, then combined multiple of them analagous to multi-layer perceptrons, evaluated on the MNIST dataset [2, 4]. We investigated the effect of adding input dropout and increasing branching factor (decreasing dendritic tree depth) and found both to have a positive effect on performance. Furthermore, we evaluated our model performance relative to MLPs of similar parameter size on binary MNIST 4-9 classification and on full MNIST multi-class classification, both as direct input and when attached to a simple convolutional neural network. Even without using the k-tree repeated attachment in Jones-Kording, our model is able to outperform parameter matched MLPs. Though all models overfit on the training data, the sparse tree structure reduces it relative to an MLP.

Our work demonstrates the potential for a dendritic tree neuron of similar parameter size to replace an MLP in a feed-forward layer, as it gives better robustness to overfitting, better performance, and potentially better interpretability due to the hierarchical tree structure. This work also exhibits the power of input non-linearity over classical perceptron architecture.

6. Limitations and Future Work

Computational optimization: The current implementation in PyTorch [7] utilizes masks on full weight matrices, which add unnecessary computation to a sparse structure. When the dendritic tree structure is deep, we cannot efficiently parallelize operations relative to parameter-matched shallower MLPs as the signal is depth-wise sequentially passed through the tree. However, the theoretical reduction in the number of operations and parameters still holds. For applicability, future work should focus on optimizing the practical computations by taking advantage of the tree structure.

Model Tuning: Our current experiments involve naively attaching the dendritic tree structure to a flattened input. Up/down-sampling is required for tree-structured vectorized computations in each layer to match the branching factor, adding rigidity. Methods for attaching tree-dendrites to input vectors to take advantage of locality structures are yet to be investigated. Furthermore, exploration can be made to automatically tune the branching factor or explore non-balanced branching as a dynamic feature.

General Applications: It remains to be discovered how well the dendritic module performs as a perceptron replacement on tasks outside of MNIST classification. Our experiments focused mainly on shallow networks; future experiments can investigate if a similar trend to current results is observed on larger, deeper, state of the art models. We also focused on testing parameter efficiency with multi-layer perceptrons against the dendritic neuron counterpart, which may result in subpar optimal accuracy. Further exploration is required to evaluate how extreme we can reduce the number of parameters in the input tree structure relative to a perceptron while preserving performance.

7. Conclusion

To augment artificial neural networks with biologically inspired input non-linearity, we created a general dendritic neuron module for tunable branching and dropout. Compared against multi-layer perceptrons, our implementation maintains theoretical parameter efficiency with better performance on tuned branching. Promising evaluation on MNIST classification independently and modularly highlights the possibility of optimizing and incorporating these structures into existing architectures for better accuracy and generalization. These results suggest input non-linearity structure as a possible direction to explore for dynamic neural networks.

8. Acknowledgements

We would like to thank Harris Chan from the University of Toronto for his encouragement and support on this project.

References

Chavlis, S. and Poirazi, P. Drawing inspiration from biological dendrites to empower artificial neural networks. *Current opinion in neurobiology*, 70:1–10, 2021.

Deng, L. The mnist database of handwritten digit images for machine learning research. *IEEE Signal Processing Magazine*, 29(6):141–142, 2012.

He, K., Zhang, X., Ren, S., and Sun, J. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE international conference on computer vision*, pp. 1026–1034, 2015.

Jones, I. S. and Kording, K. P. Might a single neuron solve interesting machine learning problems through successive computations on its dendritic tree? *Neural Computation*, 33(6):1554–1571, 2021.

Kingma, D. P. and Ba, J. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

London, M. and Häusser, M. Dendritic computation. *Annu. Rev. Neurosci.*, 28:503–532, 2005.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., and Chintala, S. Pytorch: An imperative style, high-performance deep learning library. In Wallach, H., Larochelle, H., Beygelzimer, A., d'Alché-Buc, F., Fox, E., and Garnett, R. (eds.), *Advances in Neural Information Processing Systems 32*, pp. 8024–8035. Curran Associates, Inc., 2019. URL http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2021. URL https://www.R-project.org/.

Ritter, G. X., Iancu, L., and Urcid, G. Morphological perceptrons with dendritic structure. In *The 12th IEEE International Conference on Fuzzy Systems, 2003. FUZZ'03.*, volume 2, pp. 1296–1301. IEEE, 2003.

Rosenblatt, F. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958.

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1):1929–1958, 2014.

Todo, Y., Tamura, H., Yamashita, K., and Tang, Z. Unsupervised learnable neuron model with nonlinear interaction on dendrites. *Neural Networks*, 60:96–103, 2014.

Wang, R.-L., Lei, Z., Zhang, Z., and Gao, S. Dendritic convolutional neural network. *IEEJ Transactions on Electrical and Electronic Engineering*, 17(2):302–304, 2022.

Wu, X., Liu, X., Li, W., and Wu, Q. Improved expressivity through dendritic neural networks. *Advances in neural information processing systems*, 31, 2018.

9. Appendix

9.1. Reproducibility

PyTorch implementation and experiment data linked here: github.com/zw123han/DendriticNeuralNetwork

9.2. Model Diagrams

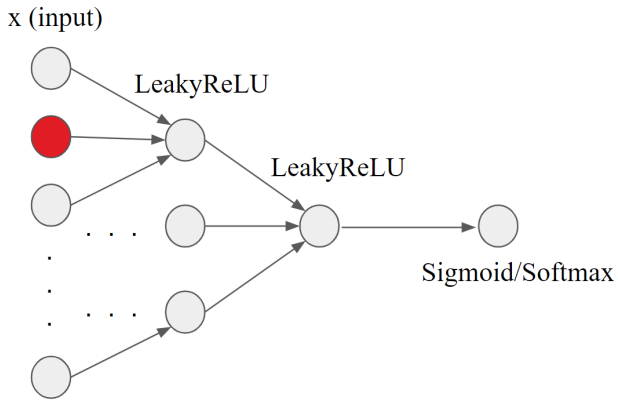

Figure 2. Architecture of a single dendritic neuron with branching $b = 3$ and flattened input size 3^k. Red represents an input dropout, which is stochastically masked at training time using the PyTorch dropout layer implementation. A LeakyReLU is applied to each layer as activation to model the input non-linearity of dendrites. The output layer, if appropriate, uses a sigmoid/softmax activation to create a probability vector.

9.3. CNN Architecture

An identical and simple CNN was attached to all classifiers. The CNN consisted of three sets of convolutions. Each convolutional layer had kernel size of 5 and a padding of 2. The number of filters in the first layer was 4, the number of filters in the second layer was 8, the number of filters in the third layer was 16. Each convolutional layer was followed by a MaxPool layer (kernel size = 2, stride = 2), a BatchNorm layer, and a ReLU activation layer.

The initial input to the CNN was a $B \times 1 \times 32 \times 32$ tensor, where B was the batch size, 1 is the number of input channels (grayscale), and 32×32 are the dimensions of the upsampled MNIST digits. Each set of convolutions and pooling cut the size of the image in half ($32 \to 16 \to 8 \to 4$). The final output of the CNN was 16 4×4 maps, which were flattened into a single 256-unit input before it was given to the dendritic classifier.

9.4. Model Parameters

Model	Weights	Biases	Total
Neuron Models 1, 4 ($b = 2$)	2046	1023	3069
MLPs 1, 3 ($h = 3$)	3075	4	3079
Neuron Models 2, 5 ($b = 4$)	1364	85	1449
Neuron Models 3, 6 ($b = 32$)	1056	33	1089
MLPs 2, 4 ($h = 2$)	2050	3	2053

Table 9. Single Neuron Experiments. Parameter computations for single-neuron models and their MLP controls. All models ran on MNIST images upsampled to 32×32 images with binary output. MLPs 1, 3 ($h = 3$) have approximately the same number of parameters as Neuron Models 1, 4 ($b = 2$), and thus serve as controls for these models. Similarly, MLPs 2, 4 ($h = 2$) are the controls for Neuron Models 2, 3, 5, 6 ($b = 4, 32$).

Model	Weights	Biases	Total
MLNs 1, 4 ($b = 2$)	20,460	10,230	30,690
MLPs 1, 4 ($h = 30$)	31,020	40	31,060
MLNs 2, 5 ($b = 4$)	13,640	850	14,490
MLPs 2, 5 ($h = 14$)	14,476	24	14,500
MLNs 3, 6 ($b = 32$)	10,560	330	10,890
MLPs 3, 6 ($h = 11$)	11,374	21	11,395

Table 10. Multi-Neuron Single-Layer Experiments. Parameter computations for multi-neuron single-layer models (MLNs) and their MLP counterparts. All models ran on MNIST images upsampled to 32×32 images, for a total 1024 inputs. Each model had 10 outputs. MLPs 1, 4 ($h = 30$) have approximately the same number of parameters as MLNs 1, 4 ($b = 2$), and thus serve as controls for these models. Similarly, MLPs 2, 5 ($h = 14$) are the controls for MLNs 2, 5, ($b = 4$), and MLPs 3, 6 ($h = 11$) are the controls for MLNs 3, 6 ($b = 32$).

Model	Weights	Biases	Total
CNN-MLNs 1, 4 ($b = 2$)	5100	2550	7650
CNN-MLPs 1, 4 ($h = 29$)	7714	39	7753
CNN-MLNs 2, 5 ($b = 4$)	3400	850	4250
CNN-MLPs 2, 5 ($h = 16$)	4256	26	4282
CNN-MLNs 3, 6 ($b = 16$)	2720	170	2890
CNN-MLPs 3, 6 ($h = 11$)	2926	21	2947

Table 11. CNN Experiments. Parameter computations for CNN classification models. All models ran on MNIST images upsampled to 32×32 images, for a total 1024 inputs. Each classifier was attached to the same CNN (described in Section 7.4), which output a 256-dim tensor. Thus, each classifier had 256 inputs and 10 outputs. CNN-MLPs 1, 4 ($h = 29$) have approximately the same number of parameters as CNN-MLNs 1, 4 ($b = 2$), and thus serve as controls for these models. Similarly, CNN-MLPs 2, 5 ($h = 16$) are the controls for CNN-MLNs 2, 5, ($b = 4$), and CNN-MLPs 3, 6 ($h = 11$) are the controls for CNN-MLNs 3, 6 ($b = 16$).