



AFRL-RI-RS-TR-2024-025

UNDERSTANDING/APPLYING NEURAL MODELS TO NEW DOMAINS

GRIFFISS INSTITUTE

MARCH 2024

FINAL TECHNICAL REPORT

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

STINFO COPY

**AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE**

NOTICE AND SIGNATURE PAGE

Using Government drawings, specifications, or other data included in this document for any purpose other than Government procurement does not in any way obligate the U.S. Government. The fact that the Government formulated or supplied the drawings, specifications, or other data does not license the holder or any other person or corporation; or convey any rights or permission to manufacture, use, or sell any patented invention that may relate to them.

This report is the result of contracted fundamental research deemed exempt from public affairs security and policy review in accordance with SAF/AQR memorandum dated 10 Dec 08 and AFRL/CA policy clarification memorandum dated 16 Jan 09. This report is available to the general public, including foreign nations. Copies may be obtained from the Defense Technical Information Center (DTIC) (<http://www.dtic.mil>).

AFRL-RI-RS-TR-2024-025 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

FOR THE CHIEF ENGINEER:

/ S /

NATHAN R. MCDONALD
Work Unit Manager

/ S /

NICK P. KOWALCHUK
RIT Chief Engineer
Computing & Communications Division
Information Directorate

This report is published in the interest of scientific and technical information exchange, and its publication does not constitute the Government's approval or disapproval of its ideas or findings.

REPORT DOCUMENTATION PAGE

1. REPORT DATE		2. REPORT TYPE		3. DATES COVERED	
MARCH 2024		FINAL TECHNICAL REPORT		START DATE AUGUST 2021	END DATE SEPTEMBER 2023
4. TITLE AND SUBTITLE UNDERSTANDING/APPLYING NEURAL MODELS TO NEW DOMAINS					
5a. CONTRACT NUMBER FA8750-21-2-0501		5b. GRANT NUMBER N/A		5c. PROGRAM ELEMENT NUMBER 62212F	
5d. PROJECT NUMBER		5e. TASK NUMBER		5f. WORK UNIT NUMBER R37T	
6. AUTHOR(S) John Salerno					
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Griffiss Institute 592 Hangar Road STE 200 Rome NY 13441				8. PERFORMING ORGANIZATION REPORT NUMBER	
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Air Force Research Laboratory/RITB 525 Brooks Road Rome NY 13441-4505			10. SPONSOR/MONITOR'S ACRONYM(S) AFRL/RI		11. SPONSOR/MONITOR'S REPORT NUMBER(S) AFRL-RI-RS-TR-2024-025
12. DISTRIBUTION/AVAILABILITY STATEMENT Approved for Public Release; Distribution Unlimited. This report is the result of contracted fundamental research deemed exempt from public affairs security and policy review in accordance with SAF/AQR memorandum dated 10 Dec 08 and AFRL/CA policy clarification memorandum dated 16 Jan 09.					
13. SUPPLEMENTARY NOTES					
14. ABSTRACT This Final Performance Report summarizes the work accomplished and lessons learned during the original and subsequent engineering change of this agreement. Specifically for the original agreement, this effort proposed three tasks: (1) understand the External Plexiform Layer (EPL) model in greater detail; (2) train the EPL model; and (3) apply the EPL model to image recognition / object identification. The engineering change (1) further investigated various training approaches / architectures to generate hyperdimensional vectors (> 1,000 bits) as the trained output instead of classification vectors; (2) applied these new models to additional datasets; and (3) investigated issues/concerns that needed to be addressed to transfer this technology to operational problems. We also investigated several newer models/architectures designed around lifelong learning and dealing with the ever-changing real world. These newer models include Random Networks; Kernet – a Kernel-based model; Sparse High-Level-Exclusive, low Level-Shared (SHELs) and OpenCon – Open-world Contrastive Learning. Detailed discussions, results and pros/cons of each model are provided.					
15. SUBJECT TERMS Neural Models, Machine Learning, External Plexiform Layer (EPL), CNNs, Kernet, SHELs, OpenCon, Lifelong learning, Contrastive Learning					
16. SECURITY CLASSIFICATION OF:				17. LIMITATION OF ABSTRACT	
a. REPORT U	b. ABSTRACT U	c. THIS PAGE U	SAR		18. NUMBER OF PAGES 82
19a. NAME OF RESPONSIBLE PERSON NATHAN R. MCDONALD				19b. PHONE NUMBER (Include area code) N/A	

Table of Contents

List of Figures	iii
List of Tables.....	v
1.0 SUMMARY	1
2.0 INTRODUCTION.....	1
3.0 METHODS, ASSUMPTIONS, AND PROCEDURES	2
3.1 Understanding the EPL Model in Greater Depth	2
3.1.1 Training.....	3
3.1.2 Enhancements.....	3
3.2 Architectures.....	3
3.3 Learning.....	4
3.4 Robustness	5
3.5 Datasets.....	5
4.0 RESULTS AND DISCUSSION	5
4.1 The EPL Model	5
4.1.1 Enhancements to the EPL Architecture.	8
4.1.2 Summary	11
4.2 Other Network Approaches	11
4.2.1 Basic CNN Models.	17
4.2.2 Integrating with Hyper Dimensional Vectors	22
4.2.3 Attention/Sharpening Function.	26
4.2.4 Using the CIFAR-10 Dataset.....	26
4.2.5 Loss Functions, Optimizers, and Learning Schedules.....	27
4.2.5.1 Adaptive SGD and WSG.....	32
4.3 Kernet – A Kernel based model.....	34
4.3.2 Computing Uncertainty.	43
4.3.3 Integrating Prediction and Uncertainty in Our GUI.....	45
4.4 Sparse High-Level-Exclusive, Low-level-Shared (SHELS).....	47
4.4.1 SHELS Concept Overview.	47
4.4.2 Modifications Required.....	52
4.4.3 Experimental Results.....	53
4.4.4 Summary.	59

4.5 OpenCon: Open-world Contrastive Learning	59
4.5.1 Modifications made to OpenCon codebase.....	62
4.5.2 Summary.	70
5.0 CONCLUSIONS	71
6.0 REFERENCES	71
7.0 LIST OF ACRONYMS	73

List of Figures

Figure 1: Overall EPL Model Architecture	1
Figure 2: CNN/Softmax Classifier.....	6
Figure 3: Self-Organizing Feature Map (SOM).....	7
Figure 4: Results of Training SOM (Centroids on Left; Test Patterns on Right).....	7
Figure 5: Current Process.....	8
Figure 6: GUI Display	9
Figure 7: Average Vectors	10
Figure 8: Similarity Measures.....	10
Figure 9: Similarity Between Average Vectors	11
Figure 10: Single Layer Model	12
Figure 11: Model Architectures Used	13
Figure 12: Multilayer Model Architecture	15
Figure 13: Area Under the Curve (AUC).....	18
Figure 14: Measuring Spread or Separation	18
Figure 15: Cosine Similarity vs Test Number (for each class)	19
Figure 16: Cosine Similarity vs Test Class (for each class).....	19
Figure 17: Plotting Separation Measures Across the Models	20
Figure 18: Visualizing Input Space in 3-D.....	21
Figure 19: Visualizing Output Space in 3-D	21
Figure 20: Proposed New Goal Architecture	22
Figure 21: T-SNE Model of Learned Class Structure	22
Figure 22: Updated Architecture.....	25
Figure 23: Revised Overall Architecture	26
Figure 24: Results of Learned Clusters.....	27
Figure 25: Normalized Hamming Distances (after 5 and 200 epochs).....	30
Figure 26: Custom Loss Function Example.....	31
Figure 27: Accuracy of the Model	33
Figure 28: Spatial Distances over Training Epochs	33
Figure 29: LeNet-5 used for MNIST End-2-End Testing	35
Figure 30: Output Results - Softmax Model.....	36
Figure 31: Output Results - Multi-Dimensional Output	37
Figure 32: LeNet-5 used for MNIST Modular Testing	37
Figure 33: Results (Cont'd)	38
Figure 34: Compared to Previous Work - Best of the Best.....	39
Figure 35: Updated Visualization Chart.....	40
Figure 36: Confusion Matrices for our 4 training scenarios	41
Figure 37: Visualization of Output Clusters.....	41
Figure 38: Output Vectors, Correct Prediction.....	42
Figure 39: Output Vectors, Wrong Prediction.....	42
Figure 40: Distance to closet point	43
Figure 41: Computing Uncertainty	44
Figure 42: Class 0 Results.....	45

Figure 43: Results Display.....	45
Figure 44: Classification Report/Confusion Matrix.....	46
Figure 45: Revised GUI.....	46
Figure 46: Code - Computing Uncertainty	47
Figure 47: Overall Process (author's paper, page 2 in [25])	48
Figure 48: Exclusivity vs Group Sparsity	48
Figure 49: Computing Thresholds	49
Figure 50: Display of Output Values.....	49
Figure 51: Novelty accommodation via selective weight updates (author's paper, page 5 in [25])	50
Figure 52: Novelty Detection.....	51
Figure 53: Results of Incremental Learning (author's paper, page 10 in [25])	51
Figure 54: High-Level Perspective of Work Tasks	53
Figure 55: Overview of Learning Framework (author's paper, page 2 in [26]).....	60
Figure 56: Baseline OpenCon Architecture	61
Figure 57: OpenCon Architecture after Adding Layer.....	63
Figure 58: OpenCon Architecture including additional backbone network (DINO_V2)	64
Figure 59: Basic Vision Transformer Architecture	64
Figure 60: Investigating Collection Metrics	66
Figure 61: Momentum = 0.9 (Baseline).....	67
Figure 62: Momentum = 0.99	68
Figure 63: Momentum = 0.999	68
Figure 64: Momentum = 1.0	69
Figure 65: Momentum = 0.0	69
Figure 66: OpenCon CIFAR-100 Dataset.....	70

List of Tables

Table 1: Parameters Used for Training.....	6
Table 2: Large feature vector size overall accuracy	9
Table 3: Single Layer Model Results	12
Table 4: Summary of 2 Model Results.....	14
Table 5: Differences After Removal of Bounds	15
Table 6: Results of Best Model	16
Table 7: Benefits of the Extraction/Classification Layers.....	16
Table 8: Summary of Statistics for Each Model	17
Table 9: Running the BasicCNN model with various configurations.....	23
Table 10: Results of Other Models	24
Table 11: Results of Running Various Dictionary Representations	25
Table 12: Initial Results for CIFAR-10 Dataset.....	27
Table 13: Comparison of Loss Functions	28
Table 14 MSE vs MAPE.....	28
Table 15: Comparison between Analog and Bipolar.....	29
Table 16: Comparing Performance using Eager Option	31
Table 17: Performance of Various Data Access Architectures.....	32
Table 18: Results of e2e Testing	36
Table 19: Modular Architecture Results	38
Table 20: Results - Comparing Fixed Seed with Different Seeds.....	39
Table 21: Selecting the Best Locations/Value for Dropout.....	44
Table 22: Results of Adding New Model Architectures.....	54
Table 23: MNIST (Learning Rate = 0.001).....	55
Table 24: MNIST (Learning Rate = 0.0001).....	55
Table 25: MNIST (Learning Rate = 0.00001).....	55
Table 26: MNIST Detection & Accommodation Results.....	56
Table 27: MNIST Detection & Accommodation Results (Cont'd).....	56
Table 28: CIFAR10 (Learning Rate = 0.001).....	57
Table 29: CIFAR10 (Learning Rate = 0.0001).....	57
Table 30: CIFAR10 (Learning Rate = 0.00001).....	57
Table 31: CIFAR10 Detection and Accommodation.....	58
Table 32: CIFAR10 Detection and Accommodation (Cont'd)	58
Table 33: Hyper- Parameters.....	61
Table 34: Baselined Test Results (taken from paper).....	62
Table 35: Test Results	62
Table 36: Test Results (After Adding Layer)	63
Table 37: Test Results - Using DINO_V2.....	65
Table 38: Test Results - Investigating the Initialization Distribution of the Prototypes	65
Table 39: Test Results - Effect of Output Feature Size	66

1.0 SUMMARY

This Final Performance Report summarizes the work accomplished and lessons learned during the original and subsequent engineering change of this agreement. Specifically for the original agreement, this effort proposed three tasks: (1) understand the External Plexiform Layer (EPL) model in greater detail; (2) train the EPL model; and (3) apply the EPL model to image recognition / object identification. The engineering change (1) further investigated various training approaches / architectures to generate hyperdimensional vectors ($> 1,000$ bits) as the trained output instead of classification vectors; (2) applied these new models to additional datasets; and (3) investigated issues/concerns that needed to be addressed to transfer this technology to operational problems. We also investigated several newer models/architectures designed around lifelong learning and dealing with the ever-changing real world. These newer models include Random Networks; Kernet – a Kernel-based model; Sparse High-Level-Exclusive, low Level-Shared (SHELS) and OpenCon – Open-world Contrastive Learning. Detailed discussions, results and pros/cons of each model are provided.

2.0 INTRODUCTION

The External Plexiform Layer (EPL) is a relatively new neural architecture attempting to understand the way in which humans smell (i.e., mammalian olfactory bulb). Architecturally the EPL model is unique and does not look like any previously researched neural models (Deep Multi-Layer Perceptron (DMLPs), Convolutional Neural Networks (CNNs), Long Short-Term Memories (LSTMs), Self-Organizing Maps (SOMs), etc.) Learning is based on local spike timing-dependent plasticity (STDP) rules and the resultant prediction or output is retrieved over sequential gamma-breadth packets of spiking activity. Figure 1 (designed/developed by Iman and Cleland [1]), provides an overview of the architecture.

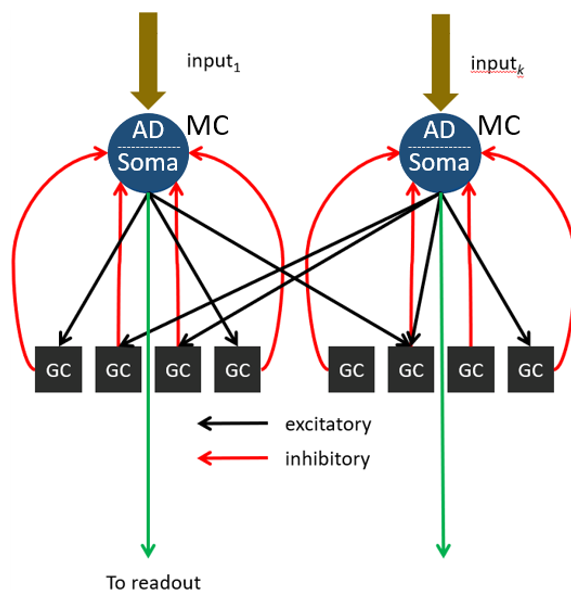


Figure 1: Overall EPL Model Architecture

The Model has two types of neurons, Mitral (MC) and Granule (GC) cells. The MC neuron contains two components: (1) Apical Dendrite (AD) and (2) Soma. Inputs are entered through the MC via the AD and sent to the Soma. There are as many MC cells as there are elements in the input vector. Each MC is connected to several GC neurons (in the author's example they use a 72-input vector (72 MCs) and 5 GCs for each MC). The network is not fully connected like other models but only partially – approximately 20% randomly selected. Weights are updated via STDP rules and support both inhibitory and excitatory connections.

Unlike many of the previous neural architectures that use a single training technique (Gradient Decent or straight STDP), EPL training is composed of two separate processes: (1) A permissive (or excitatory) part lasting for 16 timesteps and (2) an inhibitory part lasting for 24 timesteps– for a total of 40 timesteps per cycle. During the excitatory part (or feed-forward), the AD in each MC receives its corresponding input (encoded using a table look-up) and generates a corresponding spike initiation when a spike activation is crossed. The Soma is excited by spike initiation events in the AD and inhibited (or feed-back) by spikes evoked in the GCs. After propagating a spike, the soma is not permitted to spike again during the excitatory epoch. At the end of the excitatory epoch, both the AD and soma compartments are reset to zero. During the inhibitory process inhibitory synapses exhibit one of three functional states: (1) a default state of the synapse (an inactive state), exerts no effect on the MC; (2) when a spike was evoked in the GC, the synapse transitions to an inhibitory blocking state, which is maintained for a given period of time – at the end of this state, the synapse transitions to a release state for one timestep, during which it generated a unit level of excitation and then resumes an inactive state; or (3) propagates a spike when the sum of the excitation and inhibition generated by its apical dendrite and by the synapses is positive. After spiking once, the MC Soma is not permitted to spike again for the duration of that gamma cycle.

During training, each “known” odor (or sniff) is presented to the model, one odor at a time (referred to as one-shot training). That is, the first odor is processed through the network several times (each increment is referred to as a gamma cycle and the total number of cycles is set to 5 by the author) and the weights are updated after each cycle. Once completed, the next training pattern is entered. In the single odor example, the original training example is perturbed to generate 10 test samples by randomly changing 60% of the elements. Plasticity is disabled during testing.

Our goal under this effort was to look at how the EPL model can be used to understand new datasets, support hyper-dimensional output vectors and deal with new data/categories over time.

3.0 METHODS, ASSUMPTIONS, AND PROCEDURES

In this section, we discuss the specific activities we performed to accomplish the overall objectives of this effort while Section 4 discusses the actual results.

3.1 Understanding the EPL Model in Greater Depth

The EPL model is open source. Previous work with the EPL model included porting the python code provided under GitHub to Python 3, the ability to save / load trained models and provide a command line interface. The EPL model is unique and offers potentially a new view in neural

models and their ability to classify/categorize objects (whether an odor or possibly an image). The problem is that the examples presented thus far are rather simple. It is touted that such a model can incrementally learn without forgetting previous knowledge. It is this ability that we investigated.

We investigated how sensitive the values are by varying such parameters as the magnitude, the distribution and overall function to see how it affects both the training and accuracy results. We also investigated how tied this encoding is to the input values. We attempted to use the model to learn the MNIST dataset. In Section 4.1 we discuss the EPL model and the work we accomplished.

3.1.1 Training

The EPL model is a classifier. Currently the model is trained on a single instance of the category (i.e., ground truth) and resolves this input as a point in multi-dimensional space. As new categories are encountered and to be learned, they each get resolved as an additional point in this space. Separation is key to being able to resolve each category but due to the current size / number of dimensions the model can easily resolve the correct category. The dependencies on this model are thus the number of dimensions versus the number of categories. The model utilizes a similarity measure between the input and learned categories (using a Jaccard Similarity Metric). For a match two conditions must exist. First it has the highest similarity measure, and this similarity has a value greater than 0.75. If it does not meet both criteria the conclusion is that there is no match. Another approach that we considered is simply a closest distance measure. Utilizing this idea, we will investigate whether there is a difference not only in training cycles but also accuracy.

After training, a test pattern is presented to it. The test pattern is compared to each of the classes by computing a similarity measure. The one with the highest similarity or closest distance is the label / class chosen. The beauty of this approach is that during training there does not have to be an exact match. Typically, in a gradient descent supervised training approach, errors are computed by comparing the training pattern to the computed pattern and weights are modified based on these errors. In several cases a single element (or more) can be orthogonal and introduce errors when using the model. In this case, we are using a distance / similarity measure, to be correct and it only means that the predicted category / value is the closest resolved category. This approach takes advantage of the large multi-dimensional space in which we are working.

3.1.2 Enhancements

Under this task we leveraged the work we did in training the EPL model and applied the model to images. The goal here was to achieve comparable results already obtained by state-of-the-art classifiers but also show the ability of this model to learn on the fly a new class without losing prior class knowledge.

3.2 Architectures

There are two approaches that we looked at: (1) more powerful models and (2) better learning algorithms. Over time it was found that the deeper (more layers) the model had the better it did. One example is the winner of the last ImageNet competition, Residual Network (ResNet). ResNet was able to achieve >95% accuracy using 1001 layers. The problem they had to overcome was that as the network depth increased, accuracy gets saturated and then degrades rapidly. To correct

this, ResNet introduced what they called “bypasses.” A second problem with deep models is that they process sequentially and not in parallel. We compared the accuracies of all three models using both SoftMax and hyper-vectors. We also looked at a set of newer models. These models include Random Networks, Kernel-based Networks, Transformer and Contrastive Models. Many of the contrastive models sit on top of existing pretrained models. For example, ResNet as previously described or newer models called transformer networks. This newer architecture adds a projection layer on top of the pretrained model (generally referred to as a back-bone network). These networks, such as OpenCon, provide much better performance at the same time dealing with unlabeled, novel patterns. In Section 4.2 we discuss several other model architectures e.g., Random Networks, Hybrid Networks and Kernel-based Network. In Section 4.5 we introduce an example of a transformer model along with a contrastive approach.

3.3 Learning

In the earlier section, while studying the nature of sigmoid activation function, we observed that its nature of saturating for larger inputs (negative or positive) came out to be a major reason behind the vanishing of gradients thus making it non-recommendable to use in the hidden layers of the network.

Learning - Yash Bohra in [2] identifies two ways in which backpropagation can fail. These include (1) vanishing and (2) exploding gradients. In the case for vanishing gradients, as the backpropagation algorithm advances downwards (or backward) from the output layer towards the input layer, the gradients often get smaller and smaller and approach zero which eventually leaves the weights of the initial or lower layers nearly unchanged. As a result, the gradient descent never converges to the optimum. On the contrary, in some cases, the gradients keep on getting larger and larger (i.e., explodes) as the backpropagation algorithm progresses. This, in turn, causes very large weight updates and causes the gradient descent to diverge.

Certain activation functions, like the logistic function (sigmoid), have a very large difference between the variance of their inputs and the outputs. In simpler words, they shrink and transform a larger input space into a smaller output space that lies between the range of $[0,1]$. In the case of the Sigmoid function, for larger inputs (negative or positive), it saturates at 0 or 1 with a derivative very close to zero. Thus, when the backpropagation algorithm comes into play, it virtually has no gradients to propagate backward in the network, and whatever little residual gradients exist keeps on diluting as the algorithm progresses down through the top layers. So, this leaves nothing for the lower layers. We have a similar issue with the hyperbolic tangent, i.e., saturation at not only +1 but also -1. Similarly, in some cases suppose the initial weights assigned to the network generate some large loss. Now the gradients can accumulate during an update and result in very large gradients which eventually results in large updates to the network weights and leads to an unstable network. The parameters can sometimes become so large that they overflow and result in NaN values.

To tackle the issue regarding the saturation of activation functions like sigmoid and tanh, we used some other non-saturating functions like ReLu and its alternatives. Some popular alternative functions of the ReLU that mitigate the problem of vanishing gradients when used as activation

for the intermediate layers of the network are LReLU, PReLU, ELU, SELU. We presented two specific examples of alternatives to our Tanh activation function (LReLU and ELU).

We also looked at several extensions that other researchers have proposed/investigated. These included the use of an attention/sharpen function (Section 4.2.3), various loss functions, optimizers and schedulers (Sections 4.2.5).

3.4 Robustness

We investigated what it will take to deploy such a capability. Several issues remain. These include associating uncertainty/certainty with the prediction, models that evolve or grow over time (transfer/incremental learning techniques) and distributed operations (e.g., federated learning) and ultimately explain ability – why is the model producing the answer it is. We investigated work under the area of life-long learning. In Section 4.4 we introduce the Sparse High-Level-Exclusive, Low-level-Shared (SHELs) model. This work specifically looks at the life of a model after initial training. How does one identify the data/class is new and how does one update the model to learn it. As discussed above we also introduce a contrastive model which attempts to deal with real-world data (labeled and unlabeled; seen and unseen). We also investigated work with certainty. In Section 4.3.2 we present an example of a framework that defines and attempts to produce an uncertainty value that could be used by taking advantage of dropout and multiple runs.

3.5 Datasets

Work was accomplished using the MNIST dataset [3]. The dataset contains several handwritten digits from 0 to 9. This dataset is relatively simple compared to recognizing objects within a picture. MNIST images are black and white (a single channel).

We also worked with several other datasets that are more challenging. These included images of objects like CIFAR-10, CIFAR-100 [4] and ImageNet [5]. These datasets provide the images in color (use three bands or channels (RGB)) and slightly larger in size (32x32). CIFAR-10 classifies the images into 10 categories while CIFAR-100 breaks each of the 10 categories down into 10 sub-categories. CIFAR-10 and CIFAR-100 contain a total of 60,000 labeled images (50,000 for training/validation and an additional 10,000 for testing), while ImageNet, by far is much larger than either CIFAR-10 or CIFAR-100. ImageNet is an image database organized according to the WordNet hierarchy (currently only the nouns). It is the gold standard for the community. ImageNet contains 1000 classes, 1.28 million training images, and 50,000 validation images.

4.0 RESULTS AND DISCUSSION

In this section we provide a summary of the results for the multiple tasks performed under this agreement.

4.1 The EPL Model

The initial task was to test the EPL model recognizing image objects. To support this endeavor, it was agreed upon that we would use a set of exemplars from the MNIST dataset (database of handwritten digits [0-9]). Initially the idea was to send an average image built from each category. After discussion we concluded that it would be more appropriate to take an average set of activations after performing feature extraction. The reason for this was that we could consider such image transformations as scaling, rotation, and occlusion before sending it to the EPL model.

Building an average image would not support this separation. Based on this discussion we built an initial CNN to learn the MNIST dataset. Figure 2 provides a picture of the CNN/SoftMax architecture used while Table 1 provides a summary of the hyper-parameters chosen. While there are models/architectures that achieve better overall accuracies all we wanted was a reasonable, simple model. We can implement more sophisticated models later if desired.

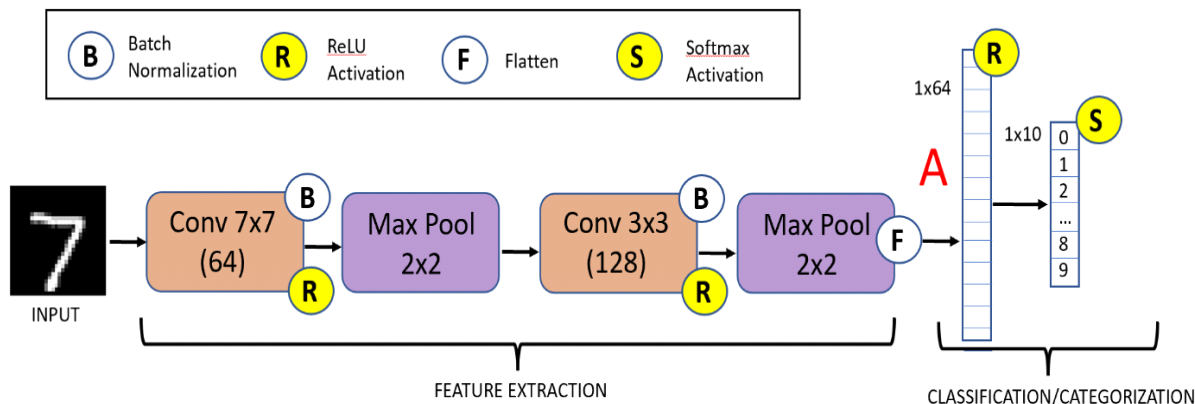


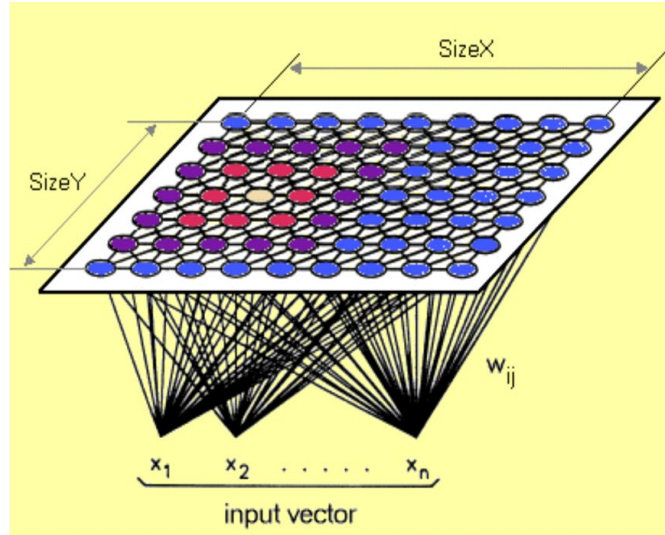
Figure 2: CNN/SoftMax Classifier

Table 1: Parameters Used for Training

PARAMETER	VALUE
Optimizer	Stochastic Gradient Decent (SGD)
Learning Rate	0.01
Momentum	0.0
<u>Regularizer</u>	L2
Weight Decay	0.0001
Batch Size	128
Epochs	25

Our CNN achieved a 99.22% accuracy against the MNIST Test set. We pulled off the feature vectors that was generated after the CNNs and just before sending it into the SoftMax layer (Labeled as “A”). We then computed the average vector for each category. We provided these 10 vectors (one for each category) to AFRL to train the EPL model.

During our in-house meetings, a question came up as to whether the average itself was the center of each cluster. To prove/disprove the concept we next built a SOM. The SOM is an un-supervised clustering technique that will take our 64-bit input vector (feature vector for each image) and collapse it to a 2-D space. SOMs use a winner-take-all learning algorithm. Figure 3 provides an architectural picture of a SOM.



<https://sites.pitt.edu/~is2470pb/Spring05/FinalProjects/Group1a/tutorial/som.html>

Figure 3: Self-Organizing Feature Map (SOM)

We trained the model using a 30x30 grid size on 4000 training patterns for 100 epochs. We then computed the centroid for each cluster. The centroid was the same as the average computed in the first part. Figure 4 provides the results of our training. Both figures display the results of training (i.e., the clusters). The figure on the left overlays the computed centroids while the figure on the right runs a handful of test samples through the model and displays how the model does classifying the image. The takeaway here is that in fact the average computed feature vector of each category (computed from the CNNs) is the center of each cluster in our SOM.

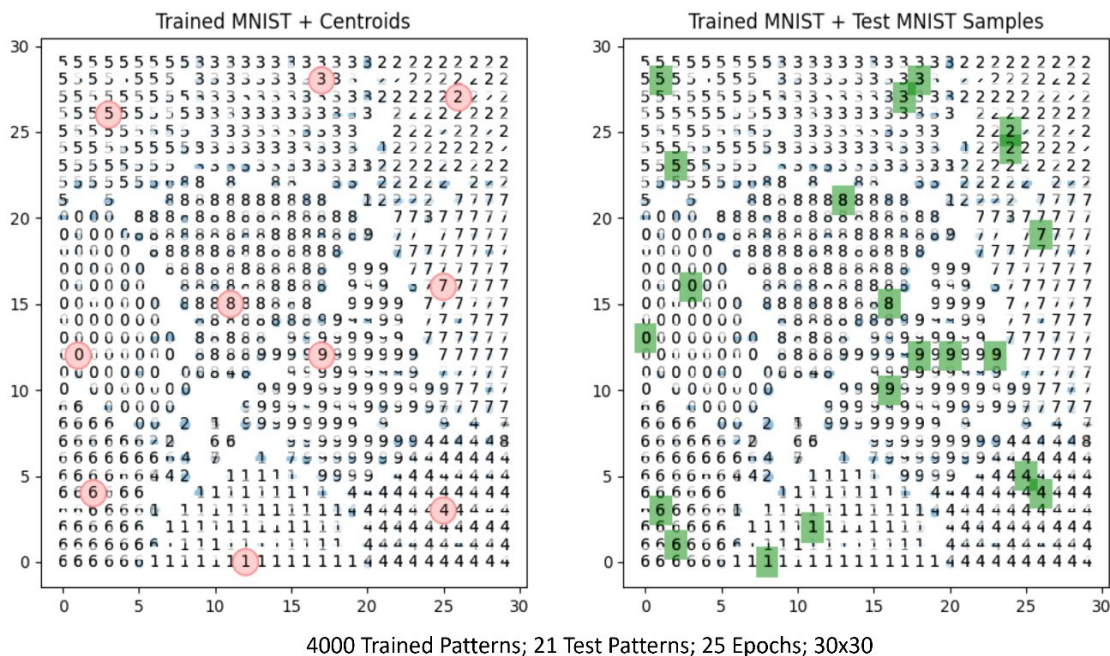


Figure 4: Results of Training SOM (Centroids on Left; Test Patterns on Right)

Figure 5 provides a summary of the overall process and what the current GUI looks like. Now that we have proven that we truly have a good set of exemplars and a baseline to begin with AFRL took them and used them to train the EPL model. In the meantime, we generated a set of sample images based off the MNIST test set. As we did this, we ran them through both models identifying those that were correctly labeled and those the model got wrong. We draw one's attention to the prediction pane. The pane displays its prediction from two models: (1) basic CNN and EPL. Notice that the CNN predicts without any doubt the correct category while the EPL model does provide a correct prediction but also provides two other possibilities: 4, 9.

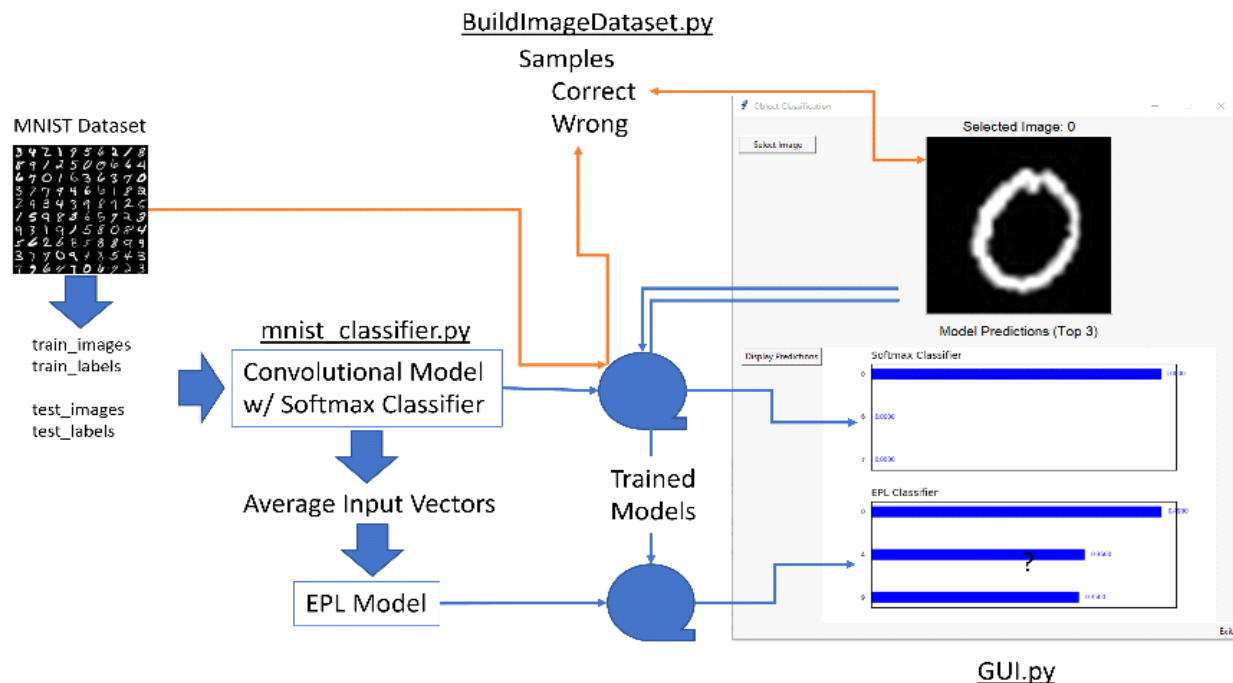


Figure 5: Current Process

While we are getting 99.22% accuracy with a rather simple CNN/SoftMax implementation we are only getting 74.66% with the EPL. This leads us to our first concern – the encoding scheme used.

4.1.1 Enhancements to the EPL Architecture.

We worked on integrating EPL and GUI code. One can now select an image and send the image through the convolutional layers and to both the SoftMax and EPL models to get their predictions. A display of the GUI is shown in Figure 6 along with loading in a sample image ('0'). In this example we received predictions from both the SoftMax and EPL classifiers of '0'.

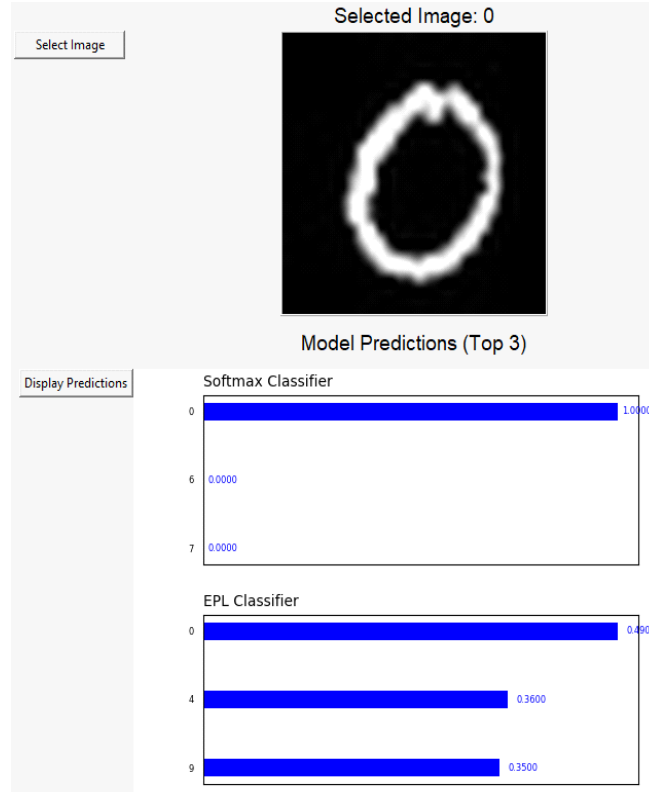


Figure 6: GUI Display

We next worked on code to look at the input vector that is generated for EPL and concerns that features could be lost due to compression. Updated code to run EPL against the MNIST test set so that we can compare results to CNN/SoftMax model.

We received additional guidance from AFRL to investigate the impact that a larger feature vector size (128/256) has on overall accuracy, computing the Jaccard similarity between target category and computed, and use of an untrained CNN model. Experimented with 128, 256, 512 and 1024 feature vector sizes and generated average vectors. Also added dropout and an additional layer to our CNN model. A summary of the results is provided in Table 2.

Table 2: Large feature vector size overall accuracy

Vector Size	Accuracy (25 Epochs)	Minimum	Maximum	Cosine Similarity	Euclidian Distance
64	99.22%	0.0	20.99	98.48%	98.29%
128	99.18%	0.0	13.89	98.48%	98.33%
256	99.22%	0.0	12.32	97.94%	97.94%
512	99.24%	0.0	7.66	97.95%	97.89%
1024	99.37%	0.0	6.42	97.42%	97.52%

EPL/Jaccard: 75.48%

Packaged up and sent 128 slot vectors to AFRL for training the EPL. Added code to produce line graph of average input vectors (Figure 7), confusion matrices for both Cosine and Euclidian similarity measures (Figure 8), and a histogram of similarity between the average vectors (Figure 9).

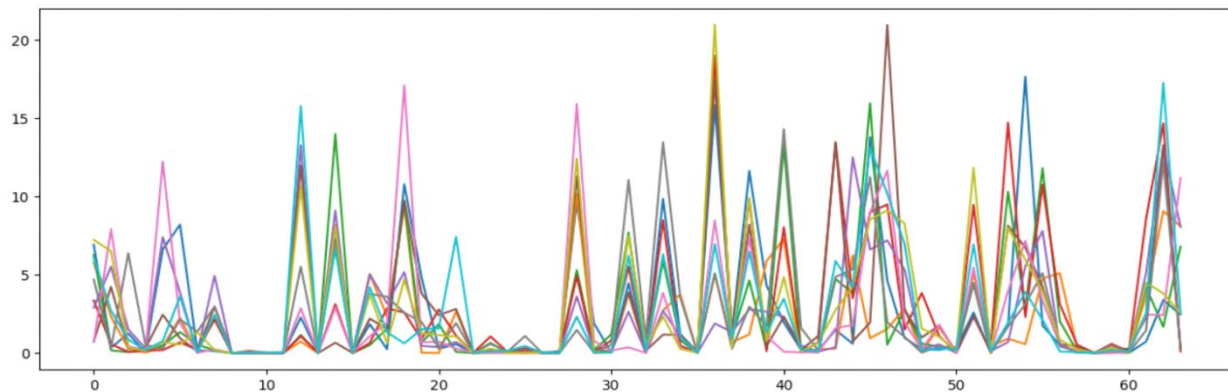


Figure 7: Average Vectors

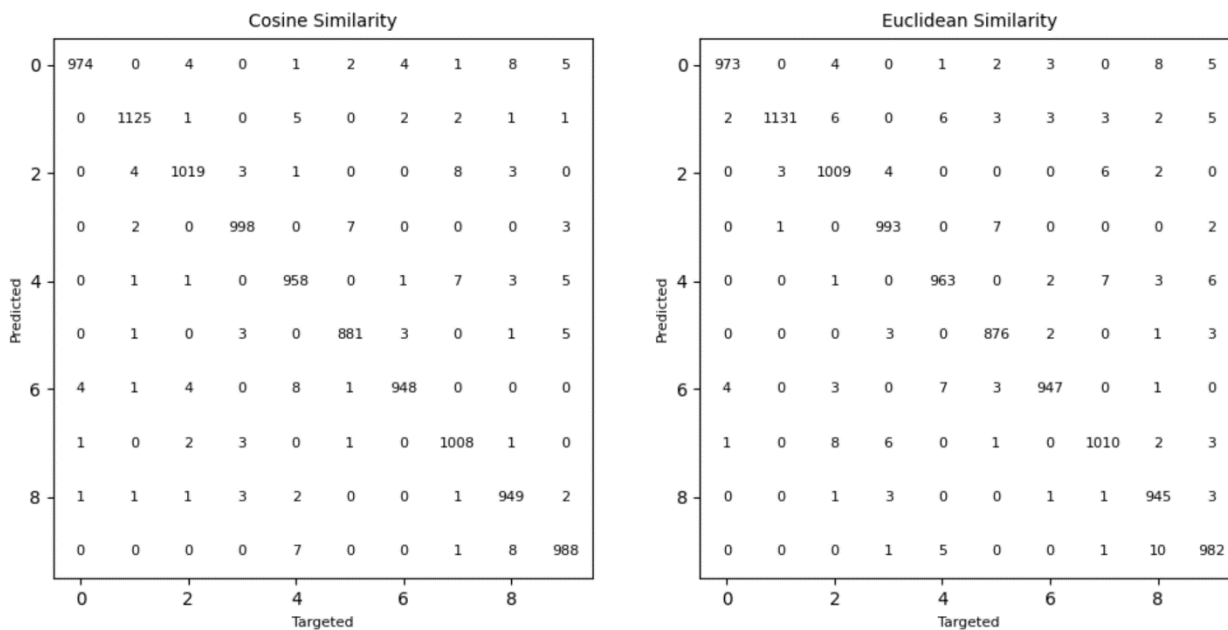


Figure 8: Similarity Measures

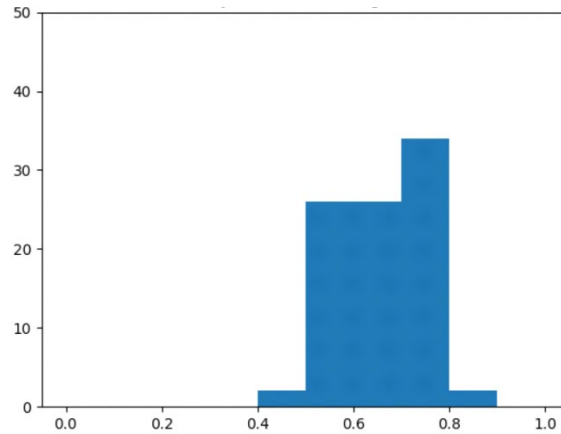


Figure 9: Similarity Between Average Vectors

4.1.2 Summary

At this point we felt that we took the EPL model as far as we could. We found that overall, the model seemed to work well on the dataset it was designed for but did not extend easily to other datasets (e.g., MNIST). We also found it to be rather slow and difficult to use. We decided at that point to look at other models/architectures.

4.2 Other Network Approaches

In this section we look at several alternative models/techniques. We begin with Random Networks where one does not need to train them to use them. We then move on to several published CNN models and end with a new model that replaces neurons with kernel functions (KerNet), Life-long learning and dealing with unlabeled, novel patterns. Integrated within the discussions are sidebars presenting several visualizations, analysis of various loss functions, optimizers, and schedulers.

For our first model we looked at a model approach based on a paper written by Langroudis, Merkel, et al. [6] which attempted to exploit randomness. We started with our basic CNN model and made several enhancements. We generated multiple runs varying the type of the random number generator, full training, or training only the dense layers, and size of the first dense layer. We also looked at the results that setting the flag trainable had on the effects on the Dropout and Batch Normalization Layers. For Random Number Generators we included: (1) Random Uniform; (2) Orthogonal; (3) Truncated Normal; (4) Glorot Normal; (5) Glorot Uniform; (6) Logistic; (7) LaPlace; (8) Gaussian and (9) Sinusoidal. We also looked at varying the bounds (for the ones we could) from $(-0.05, 0.05)$ to $(-1.0, 1.0)$. The first set of bounds were the typical ones used in most models and in many models if the initial weights exceed these values the model will not train. We explored expanding these bounds to see if there will be any improvements between the overall separation of the clusters.

We first used a single layer model to replicate the paper's results as closely as possible. Seemed that as we added a second layer, we were having problems using the author's sinusoidal function without training. The model used is shown in Figure 10. A summary of the results is provided in Table 3.

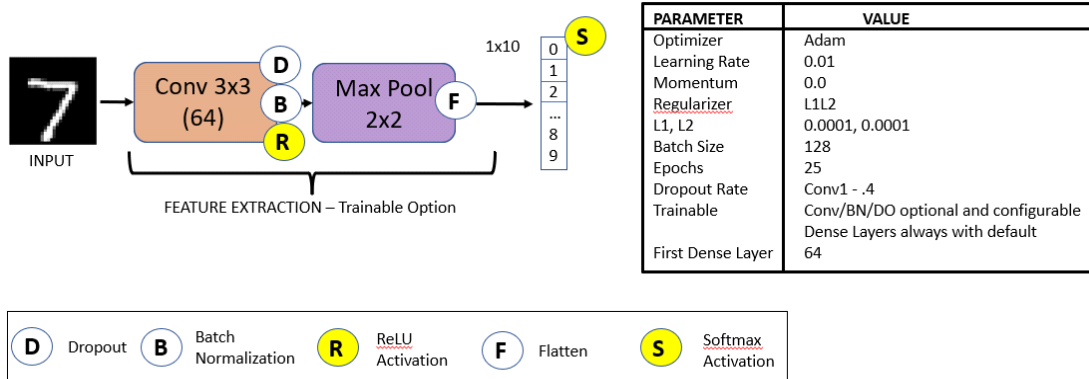


Figure 10: Single Layer Model

Table 3: Single Layer Model Results

Random Number Generator	Fully Trained (25 epochs)		Dense Only (25 Epochs)	
RandomUniform(minval=-0.05, maxval=0.05)	98.18% 98.33%	118 sec	92.98% 92.78%	79 sec
Orthogonal()	98.19% 98.21%	117 sec	95.61% 96.47%	78 sec
TruncatedNormal(mean=0.0, stddev=0.05)	98.41% 98.50%	118 sec	93.02% 93.98%	80 sec
GlorotNormal()	98.20% 98.25%	118 sec	94.09% 94.55%	84 sec
GlorotUniform()	98.19% 98.31%	115 sec	94.18% 94.10%	80 sec
Logistics: loc=0.0, scale=0.05	98.42% 97.65%	115 sec	94.84% 95.21%	84 sec
LaPlace: loc=0.0, scale=0.05	98.26% 98.23%	116 sec	94.07% 94.58%	81 sec
Gaussian: loc=0.0, scale=0.05	98.38% 98.31%	114 sec	94.65% 94.11%	80 sec
Sinusoidal: No parameters	98.53% 98.23%	121 sec	91.06% 92.84%	79 sec

Based on the results there were several observations.

- Single convolutional layer and dense layer works much better (92%) than two convolutional and two dense layers (63%).
- Results dropped greatly with multiple convolutional layers.
- Found the size of dense layer made little difference in overall results; does the number of filters make a difference?
- A dropout rate of 0.4 produced better results when training was fully turned on while a dropout of 0.5 worked best for where training was turned off on the convolutional layer.

- Sinusoidal random number generator worked best when training all layers but worked worse where we turned off the training option for the convolutional layer(s). In this case the Orthogonal random number generator produced the best results.
- On average a reduction of 37 seconds or 32% in overall training time, however, the accuracy of the untrained model dropped by 4.1/4.4% or 4.2/4.5%.
- Using the Sinusoidal Random Generator adds an initial overhead (that is much larger than the others) to the training portion since it must generate all the numbers. We are sure that one can look at a more efficient algorithm to decrease this time.

We next generated several runs by setting the trainable flag to True (Fully train the model) and False (Train only the Dense layers) for each random number generator and saved the output vectors from the flatten layer (12,544) which then becomes the input vectors for the second part (Figure 11). Like previous runs we generated average vectors for each category and computed the similarities between them. Figure 11 displays the two models that were compared while Table 4 summarizes the results.

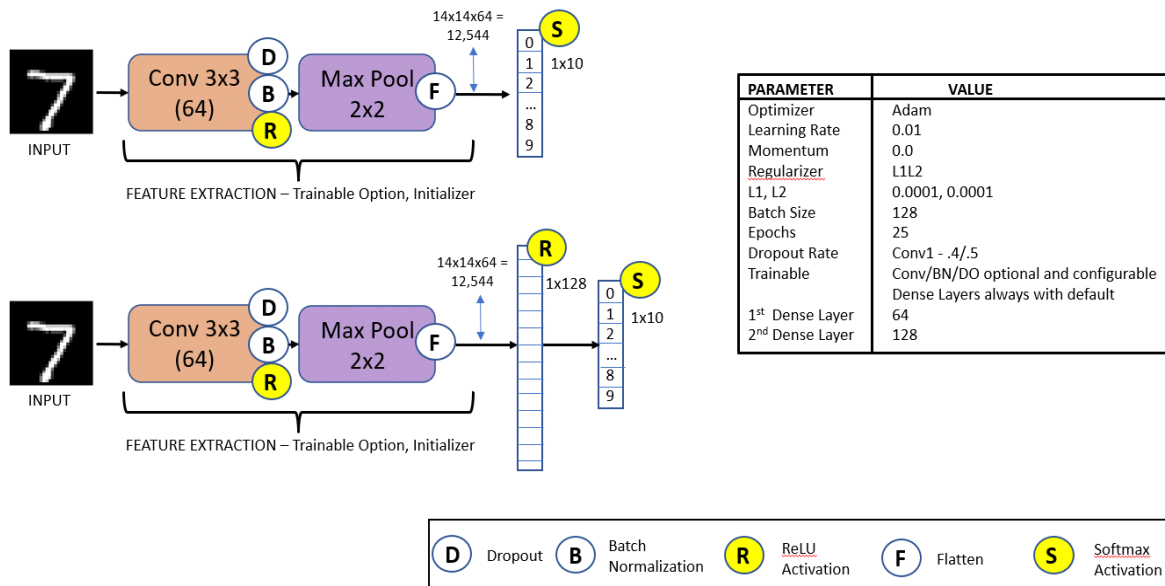


Figure 11: Model Architectures Used

Table 4: Summary of 2 Model Results

Random Number Generator	Train All Layers with 2 nd Dense Layer Added				Train only Dense Layers with 2 nd Dense Layer Added			
	Filename	Accuracy	Cosine	Euclidian	Filename	Accuracy	Cosine	Euclidian
Random Uniform	rand_mnist25_FT_RU	98.56%	89.90%	89.60%	rand_mnist25_PT_RU	94.28%	82.36%	82.74%
Orthogonal	rand_mnist25_FT_O	98.59%	90.36%	90.08%	rand_mnist25_PT_O	97.02%	82.33%	83.00%
Truncated Normal	rand_mnist25_FT_TN	98.39%	90.23%	89.83%	rand_mnist25_PT_TN	96.49%	82.76%	83.21%
Glorot Normal	rand_mnist25_FT_GN	98.33%	90.00%	89.66%	rand_mnist25_PT_GN	96.77%	82.76%	83.38%
Glorot Uniform	rand_mnist25_FT_GU	98.48%	90.12%	89.65%	rand_mnist25_PT_GU	96.56%	82.78%	82.88%
Logistics	rand_mnist25_FT_LOG	98.66%	90.57%	89.91%	rand_mnist25_PT_LOG	96.50%	81.93%	82.56%
LaPlace	rand_mnsit25_FT_LAP	98.42%	90.18%	89.95%	rand_mnsit25_PT_LAP	96.80%	82.16%	82.43%
Gaussian	rand_mnist25_FT_GUS	98.54%	90.07%	89.83%	rand_mnist25_PT_GUS	97.00%	82.05%	82.36%
Sinusoidal	rand_mnist25_FT_SIN	98.30%	89.74%	89.40%	rand_mnist25_PT_SIN	94.28%	82.16%	83.30%

Used a dropout rate of 0.5 for Convolutional Layer, Trained for 25 Epochs

Accuracy is the results after training the dense layers (in both cases) – consistent with paper

Cosine/Euclidian and Min/Max are from analyzing the activations of the flatten layer (disregards dense layers)

For trained network we dropped ~8% in accuracy while for untrained we dropped ~15%

When we trained all the layers, we found that using the Logistics Random Number Generator provided the best overall accuracy (after the dense layers and the Cosine Similarity). The best results for the Euclidian Distance were achieved using the Orthogonal random generator. For the untrained networks (where the convolutional layers weighs were not updated, i.e., only the dense layers were trained), the best overall results were achieved with the Orthogonal random generator. For Cosine Similarity it was the Glorot Uniform and Sinusoidal for the Euclidean Distance.

We investigated changing the bounds on the random numbers (-0.05, 0.05) to see if we can create greater separation between the minimum and maximum values. We modified our code to look at widening the bounds to (-1.0, 1.0) for the random numbers and investigated the use of the Adam optimizer versus SGD and re-ran our test runs. Table 5 provides a summary of the results. We found that modifying the interval (-1.0,1.0) does increase the overall accuracy rates (including Cosine and Euclidian), but not by much. Using the Adam optimizer made little difference in the overall results.

Table 5: Differences After Removal of Bounds

What happens when we lift the (-0.05,0.05) bounds from the random number generators? Let them go (-1.0 to 1.0). Might not train but we care about the output of the convolutional layer. Does it increase separation?

Random Uniform (minval = -1.0, maxval=1.0) Logistics(loc=0.0, scale=1.0)
 Orthogonal – Already (-1.0,1.0) LaPlace(loc=0.0, scale=1.0)
 TruncatedNormal(mean=0.0, stddev=1.0) Gaussian(loc=0.0, scale=1.0)
 GlorotNormal(): – Already (-1.0,1.0) Sinusoidal: – Already (-1.0,1.0)
 GlorotUniform(): – Already (-1.0,1.0)

	Accuracy	Cosine	Euclidian
Random Uniform	+3.79%	+0.20%	+0.36%
Truncated Normal	+1.94%	+0.02%	+0.10%
Logistics	+0.84%	+0.27%	+0.28%
LaPlace	+1.43%	+0.48%	+0.57%
Gaussian	+1.42%	+0.15%	+0.32%

We next looked at the effect of varying the number of layers and their sizes to increase Cosine/Euclidian similarity measures while decreasing the output of the flatten layer size. We conducted a series of runs using the same multi-layer model (Figure 12) with various random number generators. After many different runs we found that six convolutional layers followed by a dropout layer and then a max pooling layer provided us the best results. A goal in defining the model was an attempt to reduce the 12,544 output vector. Using a combination of filters and a max pooling layer we got the output vector down to 784. Table 6 provides both the specifics and overall results of our model.

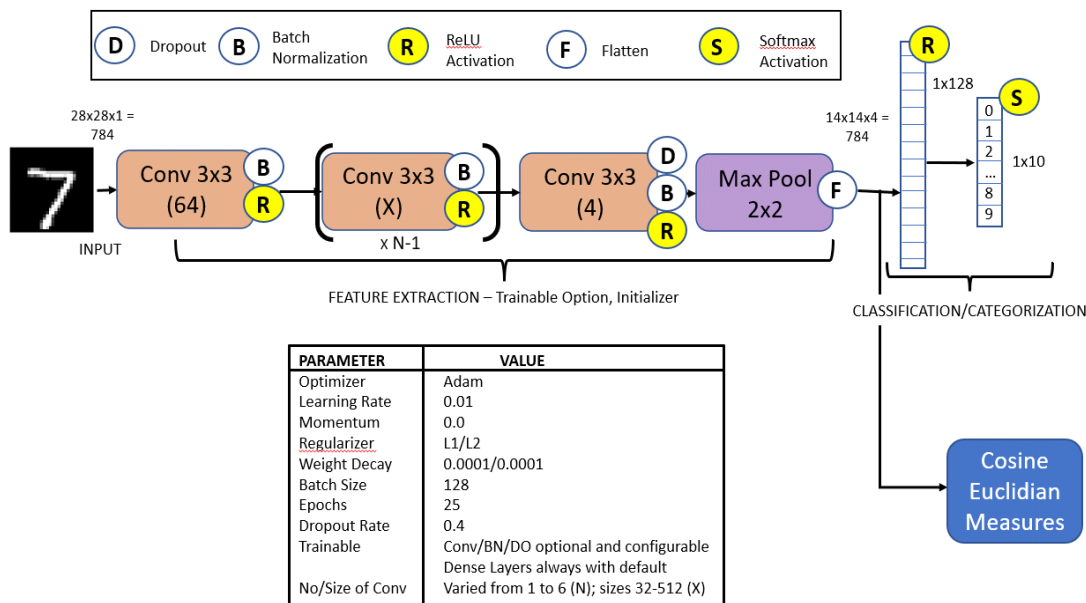


Figure 12: Multilayer Model Architecture

Table 6: Results of Best Model

	**6 Conv Layers
Model Architecture	Conv: 28x28x64 Conv_1: 28x28x128 Conv_2: 28x28x256 Conv_3: 28x28x128 Conv_4: 28x28x64 Conv_5: 28x28x4 DO (.4) MP (2x2) Flatten Dense; 128 Dense_1: 10
Flatten Layer Size	784
Accuracy	98.36%
Min/Max Values	(-0.59,17.55)
Cosine Accuracy	94.10%
Euclidian Accuracy	94.05%

Table 7: Benefits of the Extraction/Classification Layers

	Random Number Gen		Using our multi-layer model (784)		After 2 Dense Layers	
		Cosine – Inputs	Cosine Accuracy	Difference	Dense Layers Accuracy	Difference
TEST1	Random Uniform	82.52%	93.82%	11.30%	98.73%	16.21%
TEST2	Orthogonal	82.52%	93.89%	11.37%	98.64%	16.12%
TEST3	Truncated Normal	82.52%	94.25%	11.73%	98.58%	16.06%
TEST4	<u>Glorot Normal</u>	82.52%	93.84%	11.32%	98.86%	16.34%
TEST5	<u>Glorot Uniform</u>	82.52%	93.49%	10.97%	98.78%	16.26%
TEST6	Logistics	82.52%	92.74%	10.22%	98.61%	16.09%
TEST7	LaPlace	82.52%	93.77%	11.25%	98.79%	16.27%
TEST8	Gaussian	82.52%	93.22%	10.70%	98.55%	16.03%
TEST9	Sinusoidal	82.52%	94.40%	11.88%	98.83%	16.31%
	AVERAGE			11.19%		16.18%
TEST10	FULLY TRAINED	82.52%	97.60%	15.08%	98.85%	16.32%

Convolutional Layer Processing provides on average an 11.19% increase in accuracy

Adding two dense layers provide an increase on average of 16.18% and 5% using only convolutional layers

After conducting several experiments, we can say that training a model fully provides much better results. It seems that the only benefit of not training the convolutional layers is that total training time is less but at the expense of accuracy. Since training is done up front and generally only once it does not make sense to not train the full network. On average it took 40 seconds longer (for 25 epochs) to fully train our base model (single convolutional layer and single dense layer) over partially training (only dense layers). Adding additional convolutional layers caused major problems (for some reason) for the model to train when weights were not updated. The only random number generator that had a significant difference is the sinusoidal random number generator with performance. The function was not optimized, has overall poor performance and as one increased the filter sizes and/or number of filters it took exponentially longer to initialize the weights. For example, when we initialize the convolutional layer that required 128 x 256 weights it took over 41 minutes to initialize. If one were to move forward with this function one

would have to re-write it. All other random number generators performed well – unnoticeable performance impacts.

During our experiments we also found that leaving the Batch Normalization and Dropout Layers to be trainable regardless of if the convolutional layers were trainable or not had a significant difference in the overall accuracy. We found after looking into this that for dropout it did not matter. It still removed links. For the Batch Normalization layer, it did.

As described in the Kears documentation, Batch normalization applies a transformation that maintains the mean output close to 0 and the output standard deviation close to 1.

- During training (i.e., when using `fit ()` or when calling the layer/model with the argument `training=True`), the layer normalizes its output using the mean and standard deviation of the current batch of inputs.
- During inference (i.e., when using `evaluate ()` or `predict ()` or when calling the layer/model with the argument `training = False` (which is the default), the layer normalizes its output using a moving average of the mean and standard deviation of the batches it has seen during training. However, in the case of the Batch Normalization layer, setting `trainable = False` on the layer means that the layer will be subsequently run in inference mode (meaning that it will use the moving mean and the moving variance to normalize the current batch, rather than using the mean and variance of the current batch).

Based on all the above, we do not see any major benefit using random models and any further research will be done using fully trainable models.

4.2.1 Basic CNN Models.

In this section, we looked at several existing CNN models. These included: a Basic CNN – Our model re-built as a class; AlexNet [7], VGG-19 [8], ResNet-18 and ResNet-34 [9].

Table 8: Summary of Statistics for Each Model

Model	No. Parameters	Cosine – Inputs	Total Time/ Avg Time	Overall Accuracy	No. Outputs (Flatten)	Min, Max	Cosine Similarity	Euclidian
Basic CNN	845,150	82.52%	24:47 min/59.5 sec	98.55%	784	[-0.53, 21.40]	97.71%	97.79%
AlexNet	21,601,354	82.52%	09:25 min/22.6 sec	98.28%	256	[-0.91, 14.46]	98.30%	98.04%
*VGG-19	20,315,082	82.52%	29:29 min/01:12 min	98.92%	512	[-0.35, 14.18]	98.94%	97.86%
*ResNet-18	9,851,978	82.52%	17:18 min/41.5 sec	98.99%	512	[-1.25, 16.10]	98.45%	96.37%
*ResNet-34	19,967,562	82.52%	31:06 min/01:15 min	98.85%	512	[-3.99, 25.31]	98.62%	94.46%

Once these models were working, we next turned our attention to implementing additional statistics: Area Under Curve (AUC) and a plot of the computed cosine similarities for a given class. Figure 13 provides a sample of AUC. Two lines are displayed: (1) blue, the “best” we can do; and (2) red, the actual results obtained running the test set through the model. The idea behind this curve is to view those that were correctly classified at the given threshold. For example, if we

desire to classify those output vectors to be correct that were at least .6 like one of our dictionary elements, we will find that 7697 out of our 10,000 would satisfy the criteria. The goal would be to completely encompass the area under the blue curve. Those closest to 100% would be considered “better” classifiers.

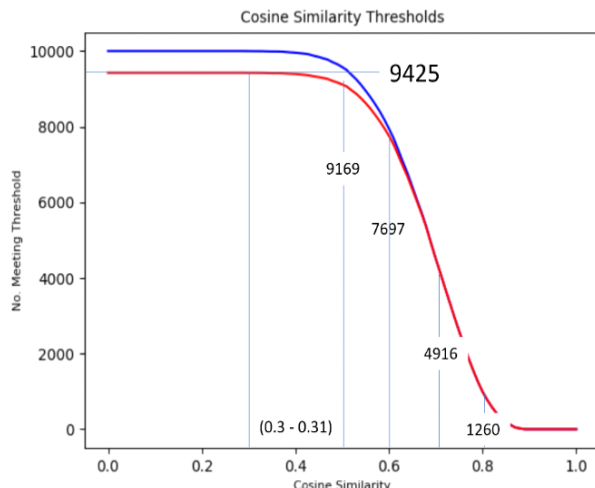


Figure 13: Area Under the Curve (AUC)

A second metric that we developed was what we called a spread metric. Looking at Figure 14, we plot the computed cosine similarity measures for each test case by class (red if it was correct blue if incorrectly identified and others as light blue). The goal is to maximize the distance between the red and light blue dots. So, in this example we compute the mean of the set of dots (red, blue, and light blue). We then compute the difference between them. This difference (e.g., red, and light blue) is what we want to maximize.

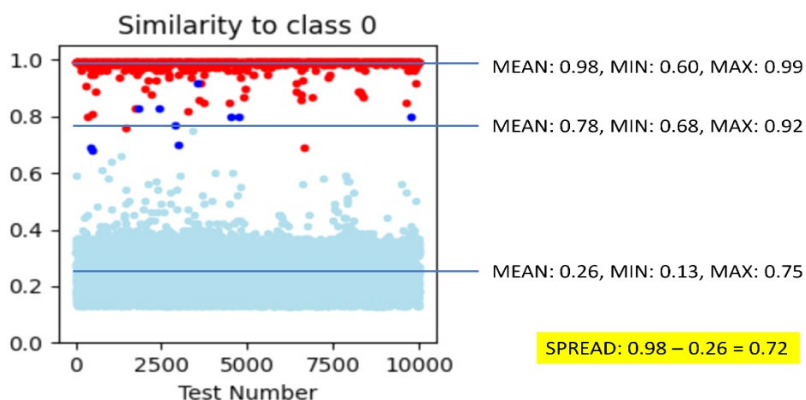
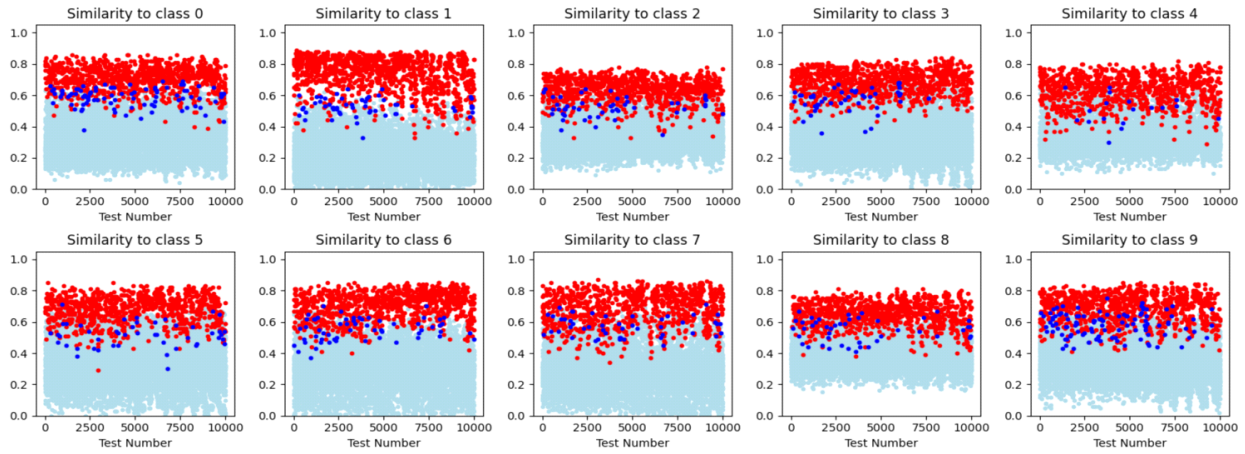


Figure 14: Measuring Spread or Separation

Figure 15 and Figure 16 provide additional sample visualizations displays.

Random Number Generator	Dense Layer Accuracy	Overall Min/Max Values	Cosine Accuracy	Euclidian Accuracy
Truncated Normal	98.58%	-0.64/20.62	94.25%	94.40%

Results of all 10,000 Test Cases

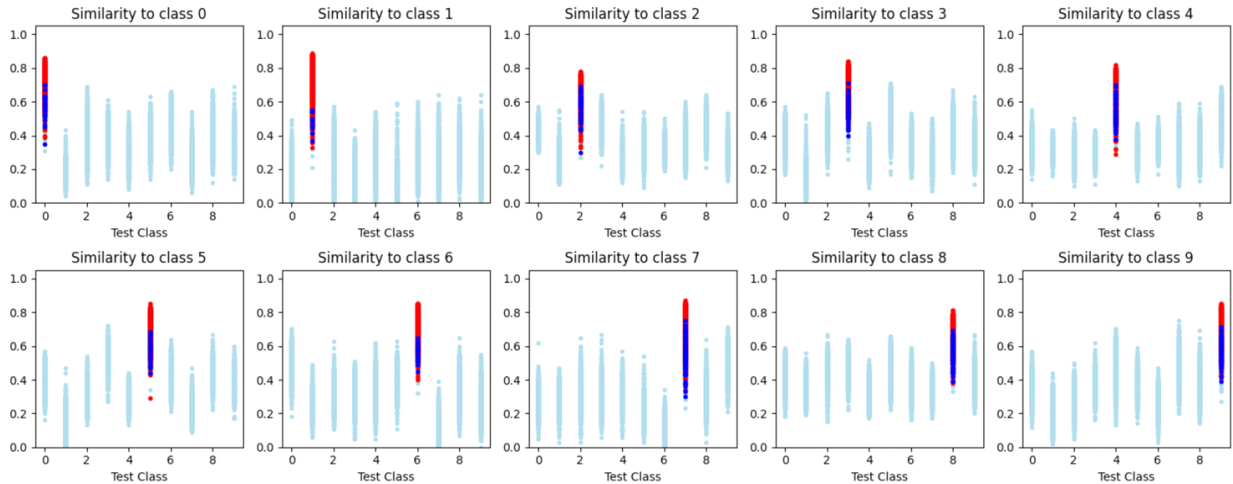


Key: Light Blue dots cosine similarities; red dots correctly predicted classes (9425); blue dots incorrectly predicted (575)
 Duplicates removed from plots

Figure 15: Cosine Similarity vs Test Number (for each class)

Random Number Generator	Dense Layer Accuracy	Overall Min/Max Values	Cosine Accuracy	Euclidian Accuracy
Truncated Normal	98.58%	-0.64/20.62	94.25%	94.40%

Results of all 10,000 Test Cases



Key: Light Blue dots cosine similarities; red dots correctly predicted classes (9425); blue dots incorrectly predicted (575)
 Duplicates removed from plots

Figure 16: Cosine Similarity vs Test Class (for each class)

We looked at the input images to see if we can compute the benefits provided by the extraction and/or classification layers. To accomplish this, we took the training set, computed an average vector for each class and then performed a Cosine Similarity calculation against each image (similar to what we did looking at the results of the output layer). We then summed up those that

we got correct and compared this value to the cosine similarities of the output layer and the overall accuracy after the dense layers.

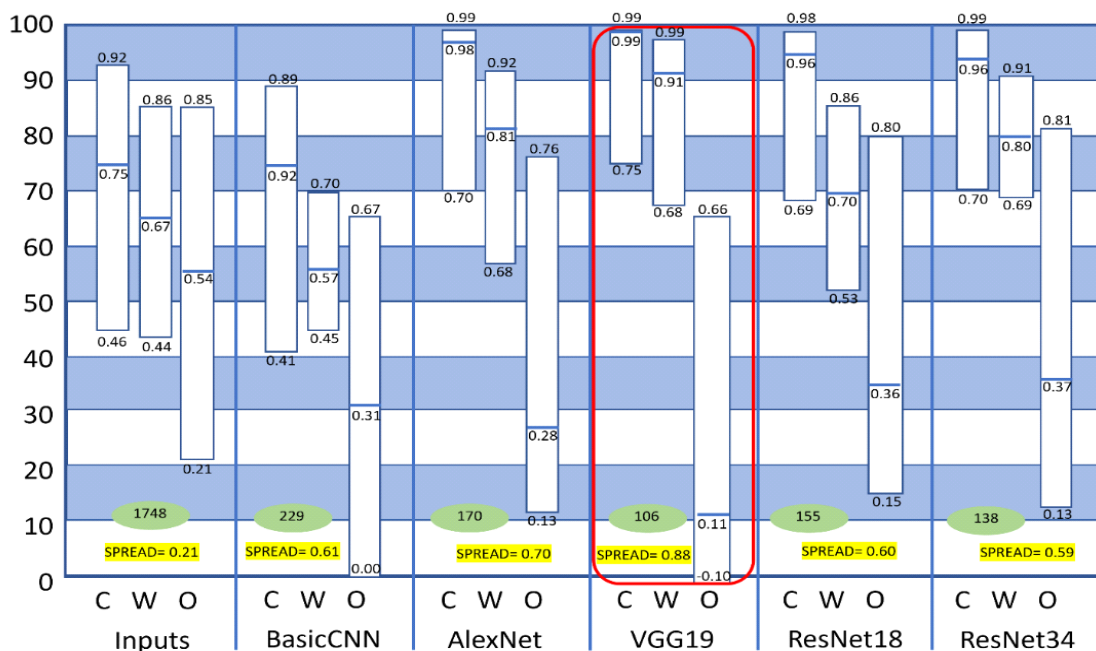


Figure 17: Plotting Separation Measures Across the Models

The VGG-19 model yielded the largest difference between the correct and other values. We can conclude from this that the VGG-19 gives us the best overall separation for this data set and does the best in separating the classes. In summary:

- AUC does not provide us with any more information to compare the various architectures, so we dropped this measure.
- Accuracy does not seem to tell us much about separation.
 - AlexNet(98.30%) & VGG-19 (98.94%) provides much better separation than BasicCNN (97.71%) and ResNet (98.45% & 98.62%).
- Spread does provide a “good” measure of comparing various architectures.

We next investigated how we might be able to take our large high dimensional output and plot it in a 2- or 3-dimensional world. Through our literature review we came across t-SNE. t-SNE which stands for t-distributed statistics neighborhood embedding system is classified as a dimension reduction technique. While Principal Component Analysis (PCA) is a very simple old technique and was widely used, nowadays t-SNE has taken its place. In all cases where PCA has limitations, t-SNE can be used. PCA preserves global structure while t-SNE preserves the local structure. It is

an iterative algorithm where at every iteration, it tries to reach a better solution. For more information see “t-SNE visualization of high dimension MNIST dataset” [10].

The beauty of t-SNE is that it has been implemented and available as a python library. We were able to implement the algorithm and apply it to both the input (Figure 18) and output vectors (Figure 19). As part of the implementation, we investigated various settings for the perplexity and training parameters. It seems that higher values for both the perplexity and training epochs provided better overall separation between clusters and better cohesiveness within each cluster. For our examples below we used a perplexity of 200 (default is 30) and set the number of training epochs to 1500 (default is 1000). The problem is that the larger these values, the longer it will take to converge. We also worked on enhancing code to plot both the centroid (average vectors) and the output vectors in the same space. Average vectors will provide the points to plot each label for the cluster.

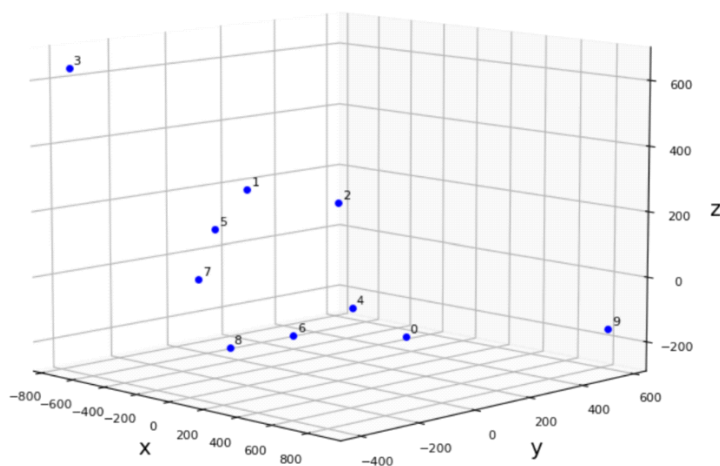


Figure 18: Visualizing Input Space in 3-D

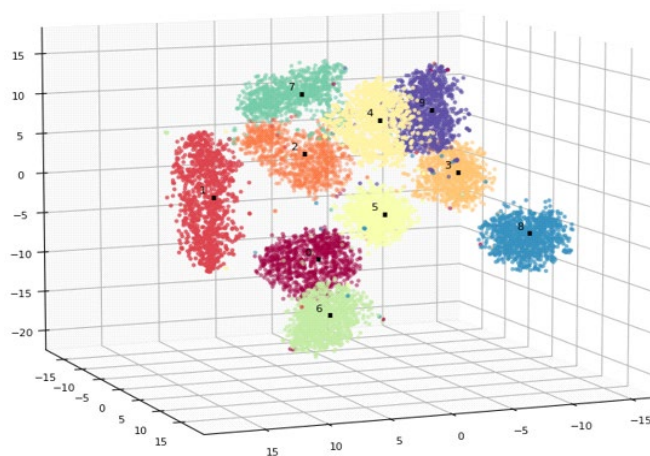


Figure 19: Visualizing Output Space in 3-D

4.2.2 Integrating with Hyper Dimensional Vectors

We reassessed where we were and believed that based on what we have seen/learned from other models it was time to revise our current architecture, Figure 20. In our updated architecture we replace the categorical outputs with hyper-dimensional vectors of d binary elements and use only one dense layer after our flatten layer with ReLU activation function. We then train using Backpropagation and at the same time use our cosine similarity as a measure of accuracy.

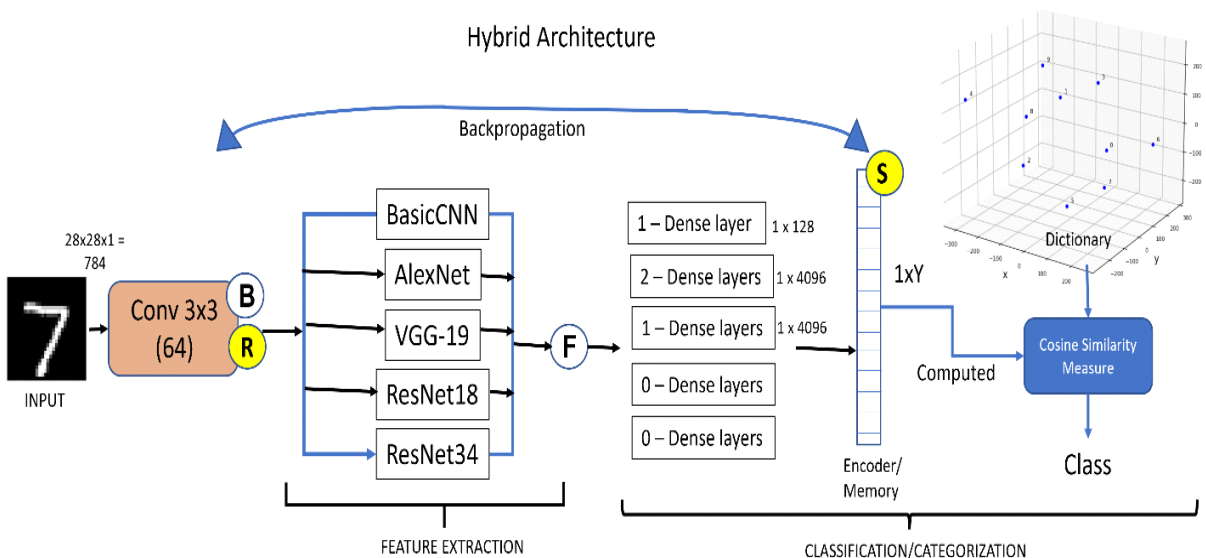


Figure 20: Proposed New Goal Architecture

Provided in Figure 21 is a plot of the results after running the t-SNE algorithm on the output of the learned MNIST dataset using the Basic CNN model. We see well-formed clusters around our center points for each class.

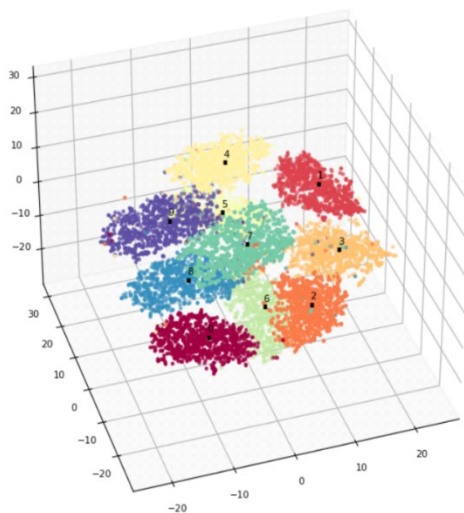


Figure 21: T-SNE Model of Learned Class Structure

Work continued to implement the features of our new hybrid architecture

- We first had to generate a random dictionary with random entries for each class (512, 1024, 2048). Initially we used binary vectors of (0, 1).
- We next implemented various similarity metrics (Cosine, Jaccard, Euclidian and Hamming).
- ...and made some minor changes:
 - Replaced SoftMax activation function with sigmoid (0,1).
 - Replaced categorical_crossentropy with mse for loss.
 - Replaced classification method.
 - Converted output vector to binary ($>0.5 == 1$).
 - Used metrics to determine accuracy.

Table 9 summarizes some of the results we obtained based on the above changes. Our best model used a dictionary size of 512 (which seemed optimum for 10 classes). In our model shown in Figure 20, we show a single inner dense layer (some models had more than one). We next asked the question whether there was a need for the additional inner dense layer(s). We ran our models with and without (Table 9). The answer in this case was yes.

Table 9: Running the BasicCNN model with various configurations

			Baseline Run		No Inner Dense Layers	
Model	Dictionary Size (X)	Epochs	Elapse Time (sec)	Accuracy (%)	Elaspe Time (Sec)	Accuracy (%)
BasicCNN	512	5	317	97.10	312	94.84
	512	10	622	98.18	620	96.03
	512	15	892	98.39	921	96.70
	512	20	1218	98.41	1222	96.84
	512	25	1498	98.38	1473	97.12
	512	50	-	-	3010	97.86
	1024	5	312	97.37	313	95.23
	1024	10	611	97.88	609	96.46
	1024	15	905	98.33	885	96.11
	2048	5	315	97.70	316	95.27
	2048	10	616	98.37	613	95.74
	2048	15	904	98.14	912	96.49

Based on our results from the above we then ran the remainder of our models. Table 10 provides these results.

Table 10: Results of Other Models

Model	Dictionary Size	Epochs	Accuracy (%)	Baseline Accuracy (%)
BasicCNN	512	20	98.41	97.71
AlexNet	1024	15	97.18	98.30
VGG-19	512	20	96.19	98.94
ResNet-18	512	20	97.84	98.45
ResNet-34	512	25	98.33	98.62

We next looked at the dictionary elements. Our original runs used a binary vector composed of 0's and 1's. We rounded the outputs to 0 or 1 (if $> 0.5 = 1$) and found both cosine and Hamming Distance gave the same results. We next investigated different ways in which we could represent the input data. We considered three alternatives:

- Bipolar (-1,1) using Cosine; converted output to bipolar.
- Bipolar (-1,1) using Hamming Distance; converted output to bipolar.
- Binary (0,1) using Cosine; output as decimal.

Table 11 provides the results of all our runs. In all cases we achieved overall better accuracy with bipolar (-1,1) and Hamming Distance than using SoftMax and cosine similarity. Additional observations include:

- It does not matter the vector size. In fact, we got worse results with larger vectors. 512 was optimum for our 10-class problem.
- Simpler models seem to outperform the large and more sophisticated ones. Not sure if we increased the size of ResNet if we get closer to Basic CNN.
- Looked at various initial learning rates, optimizers (SGD, Adam, RMS) and schedules (update after 3, 5, 10 epochs).
 - Best overall results achieved with Adam, initial learning rate=0.01 and cut in half every 5 epochs.

Table 11: Results of Running Various Dictionary Representations

Model	Dictionary Size (X)	Epochs	Cosine/Ham Binary (0,1)	Cosine Bipolar (-1,1)	Hamming Bipolar (-1,1)	Cosine Analog->Binary
BasicCNN	512	5	317/97.10	306/97.96	322/98.10	311/97.73
	512	10	622/98.18	603/97.83	609/98.08	616/98.16
	512	15	892/98.39	902/98.58 895/98.26 - .001/3 923/98.55 - .005/3 877/98.81 - .01/3 898/98.76 - .01/5	897/98.30	930/98.27
	512	20	1218/98.41	1213/98.46	1203/98.66	1195/98.34
	512	25	1498/98.38	1566/98.67 1476/98.90 - .01/5	1482/98.77 1495/98.95 - .01/5	1516/98.49 1504/98.87 - .01/5
	512	50	-	2959/96.03	2906/98.73	2976/98.96
	1024	5	312/97.37	332/97.80	314/98.01	317/97.71
	1024	10	611/97.88	635/98.32	613/98.32	604/97.91
	1024	15	905/98.33	911/98.39	891/98.52	930/98.38
	2048	5	315/97.70	315/97.82	310/97.95	315/97.76
	2048	10	616/98.37	611/98.14	612/97.85	622/97.88
	2048	15	904/98.14	892/98.56	888/98.42	979/98.02
	4096	25	-	-	1470/98.91 - .01/10	1540/98.68
	8192	25	-	-	1502/98.91 - .01/10	1557/98.85

Again, based on what we have learned we updated our overall architecture (Figure 22)

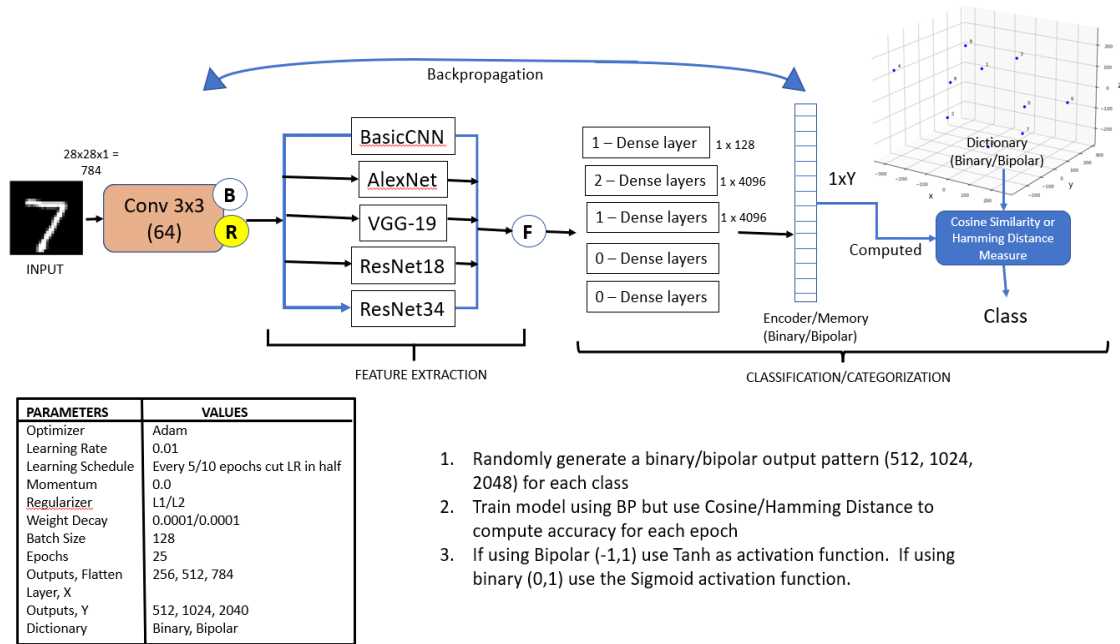


Figure 22: Updated Architecture

4.2.3 Attention/Sharpening Function.

During our literature review on hyper-dimensional architectures, we came across what the authors in their paper, “Robust High-dimensional Memory-augmented Neural Networks” [11] called a **sharpening function** which they applied to the cosine similarity metric. We implemented this function. It seemed that we got the same results with both cosine similarity and attention functions when the number of training patterns are set to the number of tests. This test proves that we have implemented the functions correctly. While interesting, at this point we were not sure about the advantage of using it.

- The paper argues that this algorithm will improve the separation margin between inter-class and intra-class similarity: 5-way (number of classes), 1-shot (number of samples)– 5.0%; 20-way 5-shot – 9.6% and 100-way 5-shot – 19.6%.
- After implementing the algorithm, we compared its results to the basic Cosine Similarity. In our case $m \times n = \text{number of test patterns} = 10,000$.
- We got identical results between cosine similarity alone and the attention function (which was expected), however, the performance was worse due to the denominator in Attention Function when using it over 10,000 test patterns.
- We did not see a reason to pursue this algorithm any further.

Based on what we have learned thus far we have **updated our overall architecture** (Figure 23).

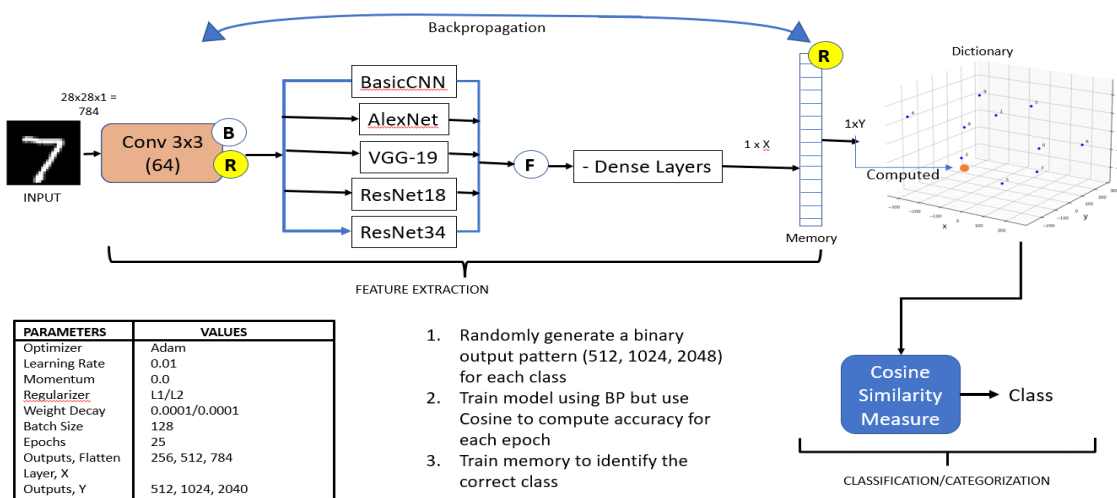


Figure 23: Revised Overall Architecture

4.2.4 Using the CIFAR-10 Dataset.

Up until now we used the MNIST dataset. The MNIST dataset is a relatively simple dataset to perform well on. We felt that we learned as much as we could from it and it was time to move on, i.e., the CIFAR10/100 datasets. As part of using the CIFAR-10 dataset we also made additional enhancements to our models. These included:

- Changed to using Mean Absolute Error (MAE) for the loss function- more robust to outliers. It is calculated as the average of the absolute difference between the actual and predicted values.
- Changed ALL dense layer activation functions to tanh. Seems that only the last layer mattered.

Table 12 provides our initial results while Figure 24 displays the output cluster chart.

Table 12: Initial Results for CIFAR-10 Dataset

Epochs	Elapse Time (sec)	Correct		Those not 100% Correct		Number Incorrect	Val_mae	Accuracy (%)
		Zero Difference	Not Identical	Maximum Difference	Average Difference			
5	206	3394	1934	164	54.44	4672	0.46364	53.28
10	396	4441	1762	163	56.81	3797	0.37893	62.03
15	593	5441	1423	152	51.28	3136	0.31506	68.64
25	1036	6371	1094	155	48.47	2535	0.25567	74.65
50	1783	7745	455	140	57.49	1800	0.18045	82.00
100	3460	8249	247	133	61.35	1504	0.15068	84.96
150	5137	8552	177	131	62.88	1271	0.12707	87.29
200	7058	8686	150	132	56.20	1156	0.11626	88.44
250	9137	8725	126	134	68.68	1149	0.11483	88.51
300	10348	8784	118	131	59.91	1098	0.11018	89.02

Model - ResNet-18 – 91.75%, 250 Epochs

Optimizer – ADAM

Learning Schedule: [0-40: .01], [41-80: .005], [81-120: .0025], [121-16, .00125], ...

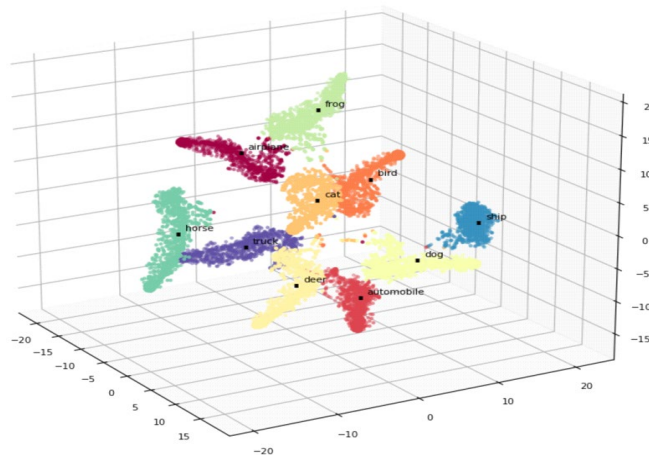


Figure 24: Results of Learned Clusters

4.2.5 Loss Functions, Optimizers, and Learning Schedules.

Does the Loss function, optimizer and learning schedule matter? In this section we looked at various Loss Functions available under the TensorFlow libraries. These include: (1) Mean Square Error (MSE); (2) Mean Absolute Percentage Error (MAPE), (3) Mean Square Logarithmic Error,

(4) LogCosh and (4) Huber. We implemented these loss functions and ran them using our BaselineCNN model and looked at the overall accuracy after 25 Epochs (Table 13). From the runs we can see that the MAPE loss function provided us with the best overall accuracy. We used the MAPE loss function on all future runs.

Table 13: Comparison of Loss Functions

Loss Function		MAE	Accuracy
Mean Squared Error	Use when you desire to have large errors penalized more than smaller ones.	0.22237	78.56
Mean Absolute Percentage Error	Consider using this loss when you want a loss that you can explain intuitively. People understand percentages easily. The loss is also robust to outliers.	0.16928	83.70
Mean Squared Logarithmic Error	Underestimates more than it does overestimates. It's a great choice when you prefer not to penalize large errors, it is, therefore, robust to outliers.	0.49614	69.45
<u>LogCosh</u>	<i>works like the mean squared error, but will not be so strongly affected by the occasional wildly incorrect prediction.</i>	0.25427	78.73
Huber	For regression problems that are less sensitive to outliers,	0.25701	78.80

We next looked at the optimizer and learning schedule. We implemented several optimizers (Adam, RMS, and SGD) and learning schedules. Table 14 provides results for our baseline model, BaselineCNN (Target is 87.48%, 150 Epochs). The results below used the Adam optimizer and a Learning Schedule: [0-40: .001], [41-80: .0005], [81-120: .00025], [121-16, .000125]. The accuracy is based on the Hamming Distance (HD). We found this combination yielded the best overall results.

Table 14 MSE vs MAPE

Epochs	MSE		MAPE (w/Learning Schedule: 40/.5)		
	val_mae	HD Accuracy (%)	val_mae	val_accuracy (%)	HD Accuracy (%)
5	0.30399	70.12	0.313	63.7	69.12
10	0.27161	74.21	0.237	70.3	76.92
15	0.23974	77.17	0.198	73.1	80.72
25	0.22237	78.56	0.168	75.3	83.72
50	0.17249	83.44	0.139	78.6	86.52
100	0.14004	86.51	0.118	79.6	88.58
150	0.12142	88.38	0.113	79.2	89.02
200	0.11794	88.79	0.115	81.0	88.94
250	0.11199	89.27			
300	0.11266	89.13			

Did better initially but eventually converged at upper epochs

The next question we addressed was whether it made a difference if we used the raw analog output vector, or one converted to bipolar (-1,1) based on if the value was less than or greater than 0. We also investigated individual class accuracies. Table 15 provides the results of a single run after 200 epochs split across each class (found at 300 epochs the results did not improve our overall accuracy). It also provides this accuracy based on whether we used analog or bipolar vectors. We did better using the raw analog output vectors based on both accuracy and overall performance.

Table 15: Comparison between Analog and Bipolar

at 200 Epochs				
	Class	Accuracy	Perfect Only	Difference
	0	91.90	90.70	1.20
	1	96.60	95.60	1.00
	2	81.40	78.60	2.80
	3	77.70	73.30	4.40
	4	89.10	87.30	1.80
	5	77.00	73.90	3.10
	6	95.10	94.10	1.00
	7	92.70	91.00	1.70
	8	92.10	90.10	2.00
	9	94.60	94.00	0.60
	Overall	88.82	86.86	1.96

We next investigated the distribution of the Hamming distances and how the distribution of correct (exact), correct due to closest (partial) and incorrect (wrong) changed over the number of epochs. In this case we used our ResNet18 model. Figure 25 provides two charts: (1) the distribution of results after 5 epochs and (2) the distribution of results after 200 epochs. As we expected, as the model learns the number of both partial and wrong results disappear and the overall number of correct ones grows. What is most interesting is the number of partial correct. These are results that would normally be incorrect in other model representations – thus our model is learning more information quicker, and training is not only correcting incorrect decisions but reinforcing those that are partial. The question becomes – how can we take advantage of this spatial correctness? Is it through a custom loss function or an optimizer that modifies weights selectively? These are a couple of questions we next attempted to address.

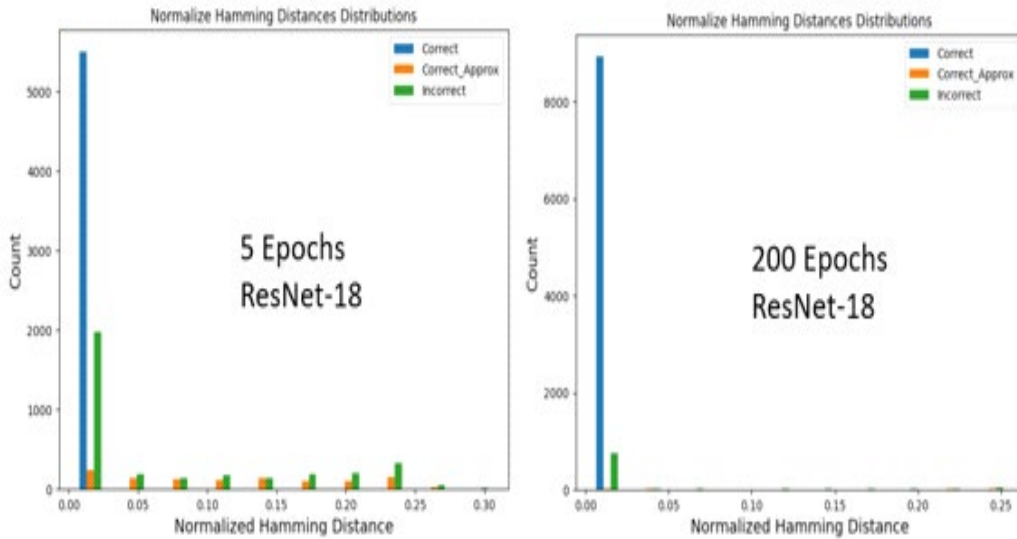


Figure 25: Normalized Hamming Distances (after 5 and 200 epochs)

Earlier we investigated various loss functions and found out MAPE, like all the others, treats all results and their losses the same. The results being an average amongst the correct, partial, and incorrect results. Is this what we want or do we want to treat this information differently? That is, do we only want to adjust weights that can be associated with the incorrect results, less attention to the partial correct results and leave the correct ones alone? We began by looking at implementing our own Loss function.

We began implementing a custom loss function. There were many examples but none very complete. The first problem we encountered was that we cannot directly access a tensor. Running tensorflow by default runs in what is called the 'fit' mode. The fit() method is fast: it runs a well-optimized, fully compiled computation graph. That's great for performance, but it also means that the code we are executing is not the Python code we have written. This can be problematic when debugging and trying to access elements within tensors.

So thankfully, there is an easy way to run our code in "debug mode", fully eagerly: pass `run_eagerly=True` to `compile()`. Our call to `fit()` will now get executed line by line, without any optimization. It is slower, but it makes it possible to print the value of intermediate tensors, or to use a Python debugger. The first question we asked is how much slower our code will be. Table 16 summarizes our results. We ran our ResNet18 model using the Adam optimizer with a Learning Schedule: [0-40: .01], [41-80: .005], [81-120, .0025], [121-16, .00125] with the run eager flag option set to True and False (None). As a result, it appears that running our model with the run eager flag = True takes approximately twice as long.

Table 16: Comparing Performance using Eager Option

Epochs	Eager=None		Eager=True		% Increase
	Elapse Time (sec)	Accuracy (%)	Elapse Time (sec)	Accuracy (%)	
5	206	53.28	429	64.40	2.1x
10	396	62.03	749	72.58	1.9x
15	593	68.64	1225	75.80	2.1x
25	1036	74.65	1834	81.14	1.8x
50	1783	82.00	4292	86.56	2.4x
100	3460	84.96	7411	89.20	2.1x
150	5137	87.29	11667	90.18	2.3x
200	7058	88.44	16092	90.75	2.4x
250	9137	88.51	17673	90.71	1.9x

We started with our code running twice as slow to begin with and wrote our own loss function. Figure 26 provides examples of two loss functions. The one on the left uses pure tensors while the code on the right does not.



```

def my_rmse(y_true, y_pred):
    # difference between true label and predicted label
    error = y_true - y_pred
    sqr_error = K.square(error)
    mean_sqr_error = K.mean(sqr_error)
    sqrt_mean_sqr_error = K.sqrt(mean_sqr_error)
    return sqrt_mean_sqr_error

def my_mse_loss(dict):
    def mseb(y_true, y_pred):
        error = K.zeros((128, 512))

        for i in range(len(y_true)):
            pred_label, _ = get_label(convert_to_bipolar(y_pred[i]), 10, dict)
            true_label, _ = get_label(y_true[i], 10, dict)
            # print('TL: ', true_label, 'PL: ', pred_label)
            if true_label != pred_label:
                error[i] = y_true[i] - y_pred[i]

        sqr_error = K.square(error)
        mean_sqr_error = K.mean(sqr_error)
        sqrt_mean_sqr_error = K.sqrt(mean_sqr_error)
        return sqrt_mean_sqr_error
    return mseb

```

Figure 26: Custom Loss Function Example

What happens when we run our loss function? After 24 hours we did not complete a single run. Why so slow? There were several issues that we uncovered and implemented several workarounds: We implemented two fixes: (1) Update a tensor vector directly; and (2) Initially we used normalized hamming distance, it took ~ 19 hours/epoch, we replace Hamming Distance Metric with the use of a KDTree for storage and query. This reduced the overall time for an epoch to ~2:45. Table 17 provides a summary of the results obtained using the various options, but 3 hours/epoch is still too slow. We next looked at multi-processing and multi-threading. We successfully created a thread and attempted to launch several of them using both a multi-threading

(multi-threading is across one CPU/core) and multi-pooling (across multiple CPU cores). Found it took ~3 times longer to run the 128 (x2) queries.

After implementing threads, we found that the multi-processing option is being hijacked by Tensorflow and believe it is trying to send multiple processes to the GPU and not trying to use the CPUs. Have not found out as to how to use the multiple CPUs. Since we have only one GPU it results in a serial process. We need to investigate other options to increase performance.

Table 17: Performance of Various Data Access Architectures

Run No.	Hamming Distance Loop (sec)	KDTree Loop (sec)	KDTree – Global Loop (sec)	ckDTree Loop (sec)
1	176.14	24.67	26.00	23.95
2	185.52	25.21	29.57	27.01
3	185.65	25.22	29.94	28.06
4	176.86	25.42	29.76	27.45
5	174.02	25.26	29.90	26.46
6	174.32	25.33	30.18	27.90
7	175.73	25.27	28.82	27.65
8	171.64	25.25	29.29	27.14
9	171.22	25.16	29.02	27.83
10	171.30	25.44	29.09	27.62
Avg/Loop	176.24	25.22	29.16	27.11
Est Time/Epoch	19:07:24	02:44:12	03:09:51	02:56:30

Estimated Time per Epoch = Avg/Loop * 50,000/128
Total Training Patterns = 50,000; Batch Size = 128

4.2.5.1 Adaptive SGD and WSG.

We have seen that over time those few patterns that the model is getting wrong continues to get closer to the wrong answer and does not self-correct. The question is if we know these patterns are wrong can we perturb the network such that the algorithm will reverse the direction of the gradients. We investigated the Adaptive SGD and ran it against our baseline. In SGD (Stochastic Gradient Decent), the learning rate value is fixed; it does not change with steps. What if we dynamically change the learning rate? It increases the speed of convergence and is more efficient on sparse data. For frequently occurring features, it decreases the learning rate and vice-versa.

$$\text{SGD} \Rightarrow w_t = w_{t-1} - \eta \frac{\partial L}{\partial w_{t-1}}$$

$$\text{Adagrad} \Rightarrow w_t = w_{t-1} - \eta'_t \frac{\partial L}{\partial w_{t-1}}$$

$$\text{where } \eta'_t = \frac{\eta}{\sqrt{\alpha_t + \varepsilon}}$$

ε is a small +ve number to avoid divisibility by 0

$$\alpha_t = \sum_{i=1}^t \left(\frac{\partial L}{\partial w_{t-1}} \right)^2 \text{ summation of gradient square}$$

We implemented the Adaptive SGD and ran it against our CIFAR-10 dataset. Figure 27 and Figure 28 provide the results.

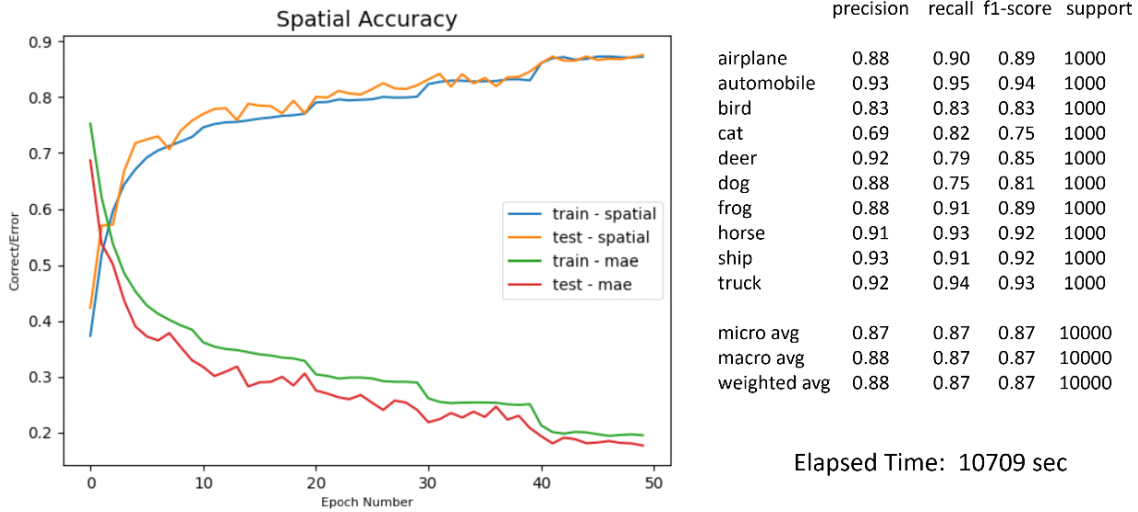


Figure 27: Accuracy of the Model

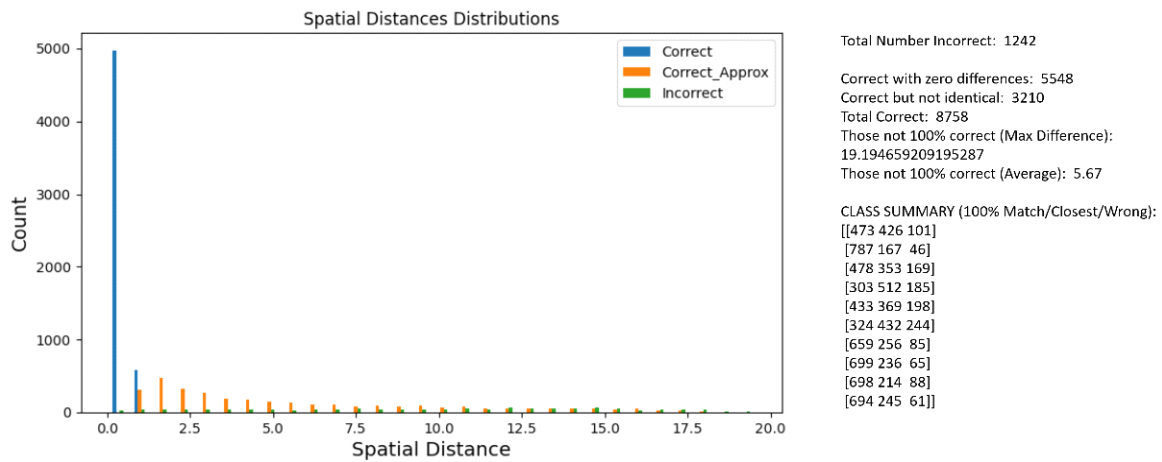


Figure 28: Spatial Distances over Training Epochs

The Adaptive SGD algorithm overall did not achieve comparable results with our current Adam Learning Algorithm (89.46% vs 87.58%). The first question was can we change the learning rate through a customized optimizer? We next went back to literature to see if we could uncover any other work in this area. We next went back to literature to see if we could uncover any other work in this area, specifically looking for any work that has looked at weighted or selective weights [11-18]. In our readings we did uncover one technique [15, 16] that attempted to dynamically modify the learning criteria. This technique is called the Weighted Stochastic Gradient (WSG) algorithm and trade-offs between the cost and monitoring performance. In WSG, a constant weight factor is used to the normalized gradient term:

$$\Delta_k = \Delta_{k-1} (1 + \mu \frac{\nabla Error_{k-1}}{2\overline{Error}_{k-1}})$$

where

$$\nabla Error_{k-1} = (Error_{k-1} - Error_{k-2})$$

$$\overline{Error}_{k-1} = \frac{Error_{k-1} + Error_{k-2}}{2}$$

After further investigation we found that it was applied to a continuous function rather than a series of individual test patterns and would not work for our problem set. We abandoned this approach.

Continue to investigate Adaptive SGD and see if we can modify the rate. If so, how will we do this?

- Do we change the learning rate or the ‘summation of the gradient square’?
- It changes ALL the weights equally – Is this what we are looking for or is weighted/selective weights where we wish to go?
-

4.3 Kernet – A Kernel based model

While we were performing our literature search, we came across a PhD dissertation that we found interesting and worth investigating a little further. This dissertation entitled: “Unifying Kernel Methods and Neural Networks and Modularizing Deep Learning” was written by Shiyu Duan [19]. The author proposes the use of kernels in place of typical neurons and a novel modular learning framework. The model is divided into two parts: (1) a Feature Extractor and (2) Classifier. The author trains the two parts as separate processes with the Feature Extractor being trained first. Recall that our previous learning approaches were based on backpropagation and the use of a training, test, and validation dataset. Leveraged from the kernel world, training of the modules requires only pairwise label information on pairs of examples and whether they belong to the same class. The full label of each individual example is not needed; neither does the system need to know the relationship among all the examples simultaneously. After training the feature extractor, he next uses its output as input to the classifier. It is then the classifier that is trained with the classical backpropagation algorithm. The Kernet code is based on Lenet-5. So, our first step was to get the code to run using the base Lenet-5 model (without the author’s extensions). We also made a few changes to bring it up to where our previous models were. These included:

- Using a multi-dimensional output vector rather than with SoftMax.
- Added code to generate a dictionary with random vectors of size 512 using bipolar values.
- Dictionary generated in example/train if it does not exist; otherwise, we read it in.
- Updated output vector size to 512 (vs 10) in LeNet model but still uses 10 classes.
- Same dictionary is used across runs (LR=0.1 (200); LR=0.01 (100) & LR=0.001 (50)).
- Used KDTree to resolve class/category and 'closest' algorithm for computing training/testing accuracy in train.py.
- Added 'mse' and 'l1' as available loss functions.
- Modified test.py to use same dictionary.
- Made output_vector_size a variable passed to the LeNet model (opt.n_out_features).
- Pass opt.loss so that we know what we are running – allowed one to run either categorical or multi-dimensional variants.
- Modified Test.py to work.
- Modified Visualization to work (modified code to remove PCA and changed n_iter=2000, perplexity=150).
- Generated multiple configuration scripts to run training, testing and visualization code.

Figure 29 displays the two architectures (top for our multi-dimensional output model and lower one for the original SoftMax architecture) we used, while Table 18 provides the results and Figure 30 and Figure 31 displays the t-SNE 3-D plots. The model was run using either Cross Entropy (XE) and Mean Square Error (MSE) for the loss function depending on if we were using a categorical or hyper-dimensional output.

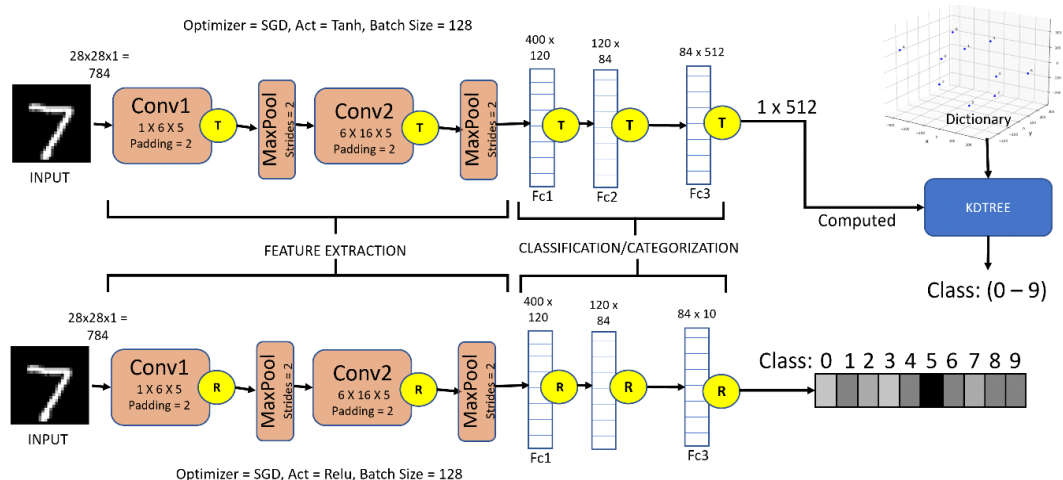


Figure 29: LeNet-5 used for MNIST End-2-End Testing

Table 18: Results of e2e Testing

Configuration Filename	No. Epochs	Act Function	Loss Function (.loss)	No. Classes	No Output Features (.n_out_features)	Learning Rate (.lr)	Total Time (Min)	Acc (%)	Actual Epochs Run
train_lenet_xe_sgd_lr0.1 (original) test_lenet_xe_lr0.1 (original)	200	Relu	XE	10 (Default)	10	0.1	36.0	98.99	200
train_lenet_xe_sgd_lr0.01 (original) test_lenet_xe_r0.01 (original)	100	Relu	XE	10 (Default)	10	0.01	19.0	99.24	168/267
train_lenet_xe_sgd_lr0.001 (original) test_lenet_xe_lr0.001 (original)	50	Relu	XE	10 (Default)	10	0.001	10.5	99.24	171/220
train_lenet_mse_sgd_lr0.1 test_lenet_mse_lr0.1	200	Tanh	MSE	10 (Default)	512	0.1	48.0	98.97	200
train_lenet_mse_sgd_lr0.01 test_lenet_mse_lr0.01	100	Tanh	MSE	10 (Default)	512	0.01	29.0	99.10	125/224
train_lenet_mse_sgd_lr0.001 test_lenet_mse_lr0.001	50	Tanh	MSE	10 (Default)	512	0.001	13.0	99.10	139/188

ReLU with Cross Entropy provided the best overall results compared to Tanh with MSE (99.24% vs 99.10%) but required many more runs/time (168 vs 125).

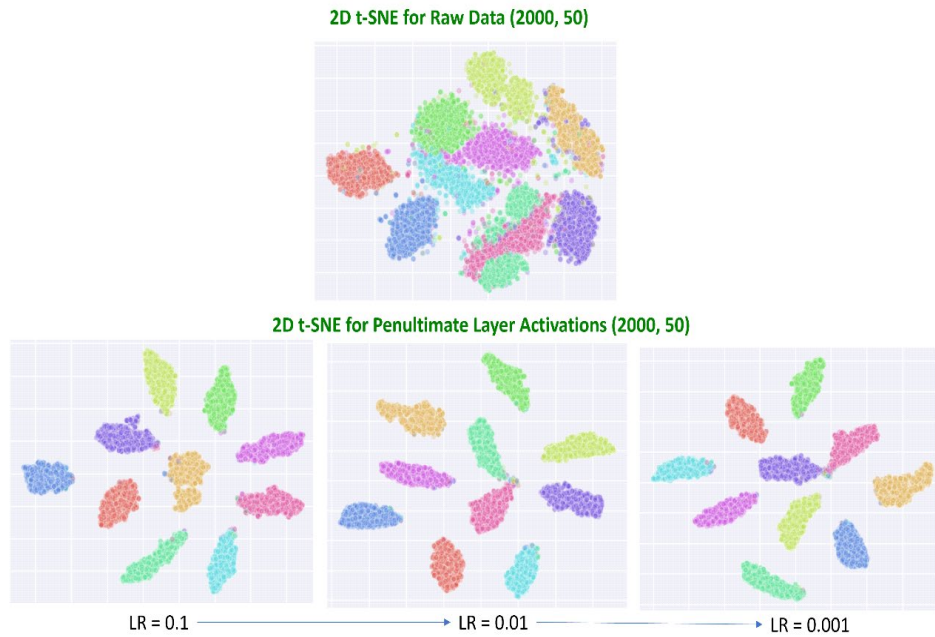


Figure 30: Output Results - SoftMax Model

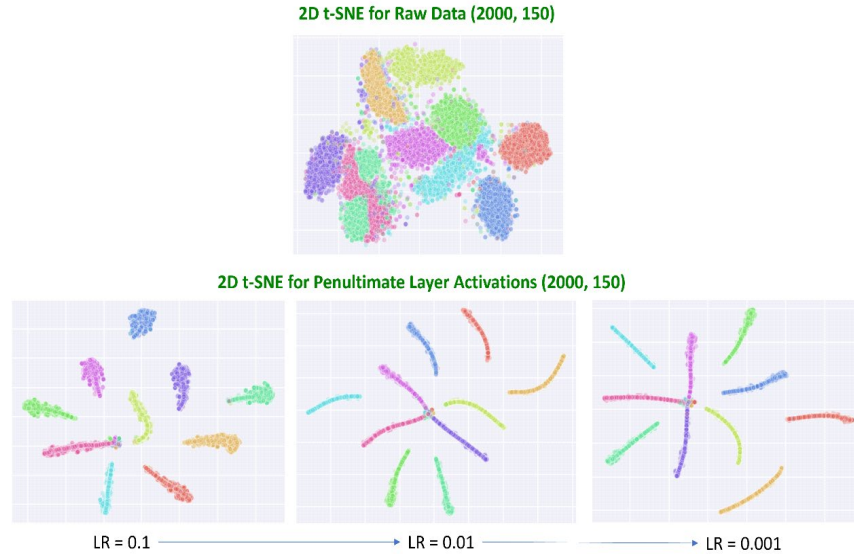


Figure 31: Output Results - Multi-Dimensional Output

We next moved on to the modular portion of the work. To accomplish this, we:

- Converted the existing Modular scripts to use configuration scripts.
- Modified modular code for training to incorporate dictionary.

We converted the back end (train_output.py) to use multi-dimensional vectors.

- Use same kernel (srs_upper_tri_alignment) with both ReLU (SoftMax version) and Tanh (multi-dimensional model).
- Looked also into srs_nmse and srs_nmse_neo and compared results.

Figure 32 provides how our models are split (2 layers) based on the kernel modular code while Table 21 provides a summary of the results.

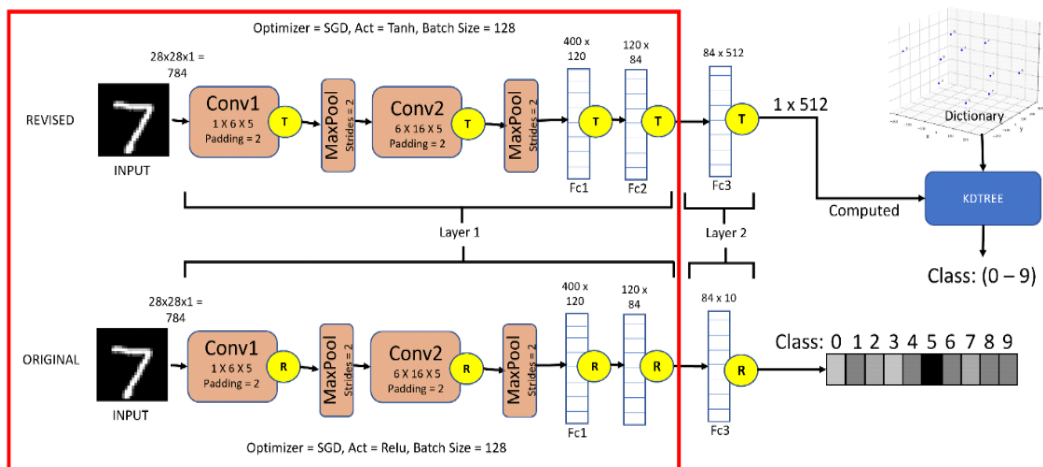


Figure 32: LeNet-5 used for MNIST Modular Testing

Table 19 shows that the best overall results were 99.30% which is a little higher than what was achieved with the e2e model using SoftMax (99.24%).

Table 19: Modular Architecture Results

Run	LR=0.1	LR=0.01	LR=0.001	Modular
[xe-seed3] Original e2e (Relu), Loss = xe	Validation: 99.24 Test: 98.99	Validation: 99.32 Test: 99.24	Validation: 99.32 Test: 99.24	N/A
[xe-seed5] Original e2e (Tanh) Loss = xe	Validation: 99.36 Test: 99.07	Validation: 99.48 Test: 99.22	Validation: 99.48 Test: 99.24	N/A
[xe-seed10] Modular Original (Relu) Loss = xe, srs_upper_tri_alignment	Validation: 98.50 Test: N/A	Validation: 98.80 Test: N/A	Validation: 98.80 Test: N/A	Validation: 99.30 Test: 99.30
[xe-seed15] Modular Original (Tanh) Loss = xe, srs_upper_tri_alignment	Validation: 58.80 Test: N/A	Validation: 59.60 Test: N/A	Validation: 59.70 Test: N/A	Validation: 99.12 Test: 99.16
[mse-seed3] Multi-Dimensional e2e (Tanh) Loss = mse	Validation: 99.06 Test: 99.30	Validation: 99.08 Test: 99.10	Validation: 99.08 Test: 99.10	N/A
[mse-seed10] Modular Multi-Dimensional (Tanh) Loss = mse, srs_upper_tri_alignment	Validation: 58.80 Test: N/A	Validation: 59.60 Test: N/A	Validation: 59.70 Test: N/A	Validation: 99.10 Test: 99.13
[mse-seed20] MDM (Tanh) Loss = mse, srs_nmse	Validation: -70.80 Test: N/A	Validation: -70.80 Test: N/A	Validation: -70.80 Test: N/A	Validation: 99.34 Test: 99.04
[mse-seed25] MDM (Tanh) Loss = mse, srs_nmse_neo	Validation: -80.00 Test: N/A	Validation: -79.90 Test: N/A	Validation: -79.80 Test: N/A	Validation: 99.00 Test: 99.20

Figure 33 plots the epoch number vs the validation accuracy for several runs (to include e2e and modular) varying the seed value. In the modular scenarios we see the overall error rate goes down using Layer 1 but as soon as we turn the model over to layer 2, we see a sharp increase in accuracy. The multi-dimensional model that did the best was ‘mse-seed20’. It reaches an accuracy of 99.34 at 137 epochs, while an accuracy of 99.48 was achieved after 202 epochs for the SoftMax with cross entropy model.

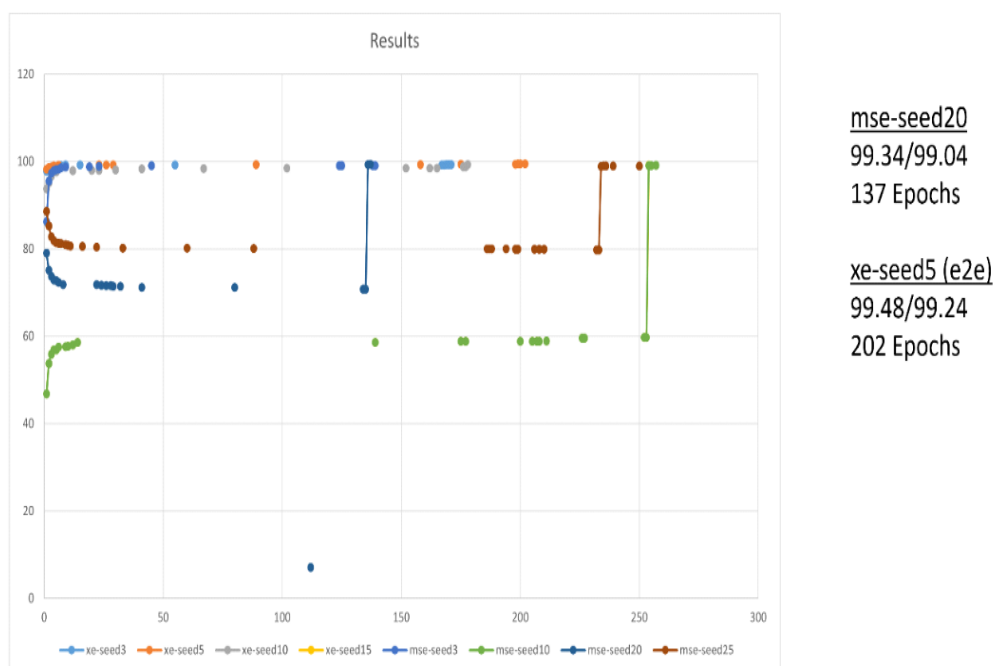


Figure 33: Results (Cont'd)

Figure 34 compares our results to previous model runs/architectures. We can see overall that both our e2e and modular model beat our previous models, but at the expense of the amount of epochs/time.

Model	Dictionary Size	Epochs	Accuracy (%)
BasicCNN	512	20	98.41
AlexNet	1024	15	97.18
VGG-19	512	20	96.19
ResNet-18	512	20	97.84
ResNet-34	512	25	98.33

<

1%

mse-seed20
99.34/99.04
137 Epochs

xe-seed5 (e2e)
99.48/99.24
202 Epochs

...and using LeNet

Figure 34: Compared to Previous Work - Best of the Best

Our first set of experiments was to look at the impact that the seed value has on the overall results. When we ran the original set, we varied the seed value for each set. Table 20 provides the results. The numbers in red provide the differences. There are differences but many of these differences are small. For comparison purposes we used a fixed random seed (= 3) for any future runs.

Table 20: Results - Comparing Fixed Seed with Different Seeds

Run	LR=0.1	LR=0.01	LR=0.001	Modular
[xe-seed3] Original e2e (Relu), Loss = xe	Validation: 99.24 (-/0.0) Test: 98.99 (-/0.0)	Validation: 99.32 (-/0.0) Test: 99.24 (-/0.0)	Validation: 99.32 (-/0.0) Test: 99.24 (-/0.0)	N/A
[xe-seed5] Original e2e (Tanh) Loss = xe	Validation: 99.36 (-/0.0) Test: 99.07 (-/0.0)	Validation: 99.48 (-/0.0) Test: 99.22 (-/0.0)	Validation: 99.48 (-/0.0) Test: 99.24 (-/0.0)	N/A
[xe-seed10] Modular Original (Relu) Loss = xe, srs_upper_tri_alignment	Validation: 98.20 (↓/0.3) Test: N/A	Validation: 98.50 (↓/0.3) Test: N/A	Validation: 98.50 (↓/0.3) Test: N/A	Validation: 99.18 (↓/0.12) Test: 99.23 (↓/0.07)
[xe-seed15] Modular Original (Tanh) Loss = xe, srs_upper_tri_alignment	Validation: 59.10 (↑/0.3) Test: N/A	Validation: 59.30 (↓/0.3) Test: N/A	Validation: 59.30 (↓/0.3) Test: N/A	Validation: 98.96 (↓/0.16) Test: 99.22 (↑/0.06)
[mse-seed3] Multi-Dimensional e2e (Tanh) Loss = mse	Validation: 99.06 (-/0.0) Test: 99.30 (-/0.0)	Validation: 99.08 (-/0.0) Test: 99.10 (-/0.0)	Validation: 99.08 (-/0.0) Test: 99.10 (-/0.0)	N/A
[mse-seed10] Modular Multi-Dimensional (Tanh) Loss = mse, srs_upper_tri_alignment	Validation: 59.10 (↑/0.3) Test: N/A	Validation: 59.30 (↓/0.3) Test: N/A	Validation: 59.30 (↓/0.3) Test: N/A	Validation: 98.94 (↓/0.16) Test: 99.19 (↑/0.06)
[mse-seed20] MDM (Tanh) Loss = mse, srs_nmse	Validation: -71.4 (↓/0.34) Test: N/A	Validation: -71.10 (↓/0.30) Test: N/A	Validation: -71.10 (↓/0.34) Test: N/A	Validation: 99.02 (↓/0.32) Test: 99.13 (↑/0.09)
[mse-seed25] MDM (Tanh) Loss = mse, srs_nmse_neo	Validation: -80.01 (↓/0.01) Test: N/A	Validation: -79.90 (-/0.0) Test: N/A	Validation: -79.90 (↓/0.10) Test: N/A	Validation: 98.92 (↓/0.08) Test: 99.17 (↓/0.03)

We next added category labels to our charts. Figure 35 shows an example of the new display with the labels.

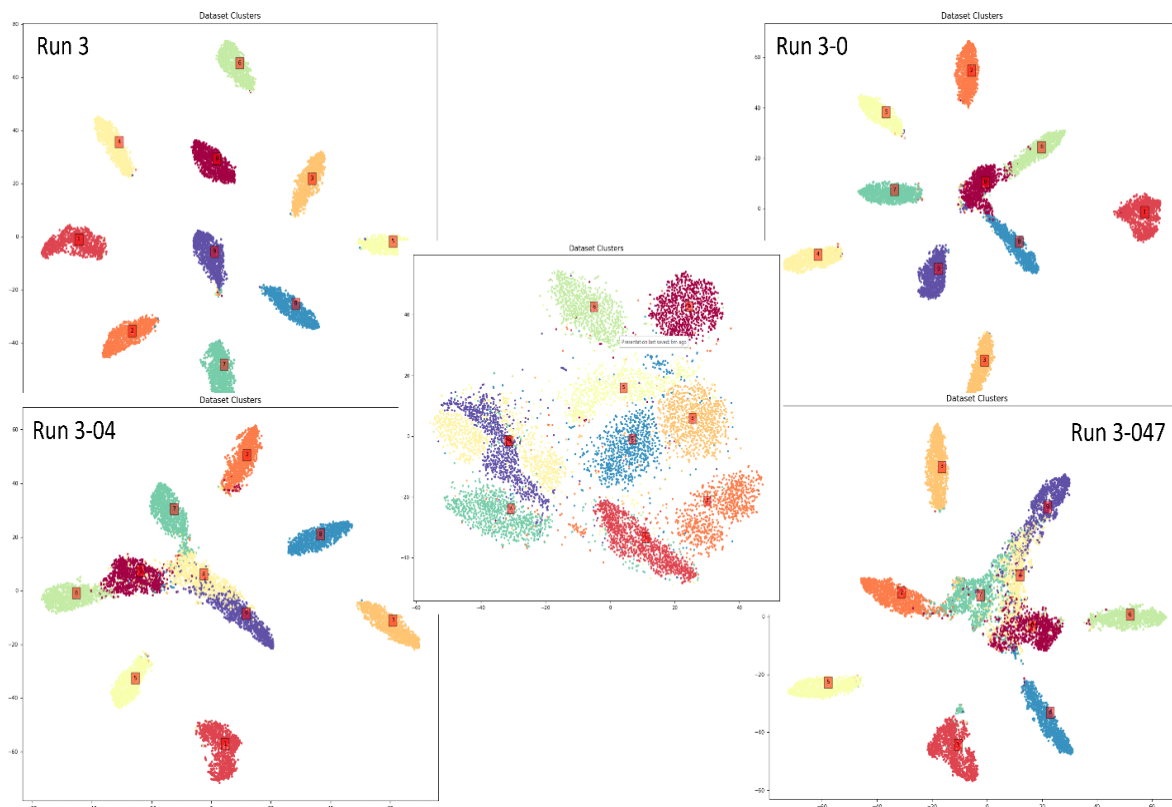


Figure 35: Updated Visualization Chart

After updating the visualization utility, we next investigated the impact that partial training has on the overall results. In this scenario we trained the model on only a subset of the classes. For example, we trained on 7 of the 10 classes and then tested on all 10 to see if/how well the model does. We updated the code to allow for an exclusion list, i.e., a list of classes that would be excluded from the training set and added a confusion matrix output. Figure 36 provides the results of running our model with four different training scenarios: (1) all 10 classes; (2) 9 classes with the class 0 removed; (3) 8 classes with the classes 0 and 4 removed; and (4) 7 classes with the classes 0, 4 and 7 removed. As expected, the row/column combination shows a significant number of incorrect results.

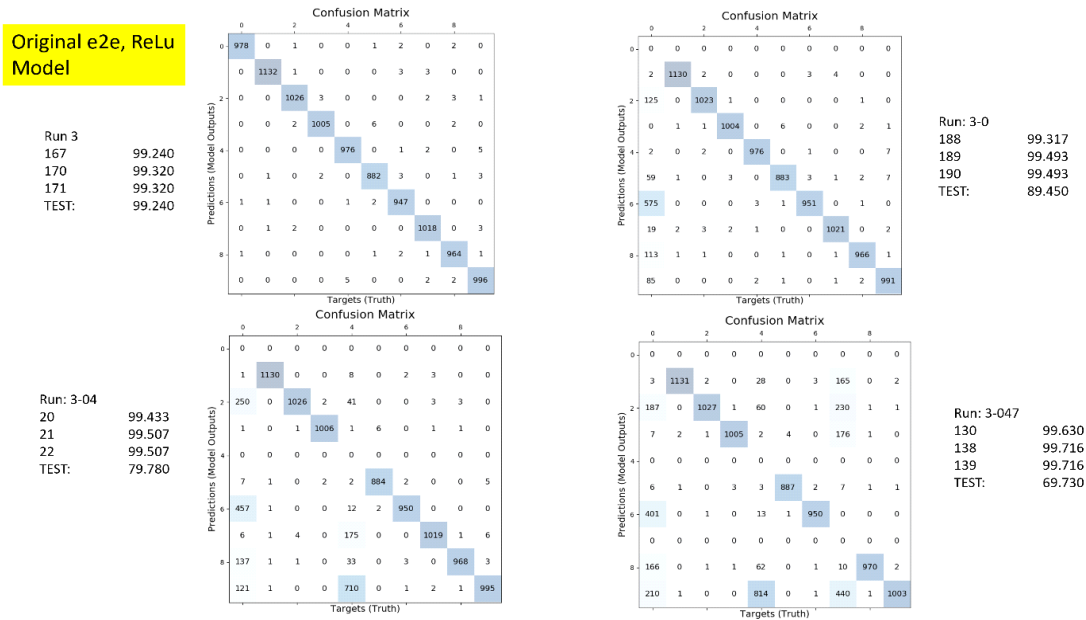


Figure 36: Confusion Matrices for our 4 training scenarios

Figure 37 provides a visual summary of our outputs. The center display shows the data, i.e., what we are starting with, while the four charts around the outside show the results of training.

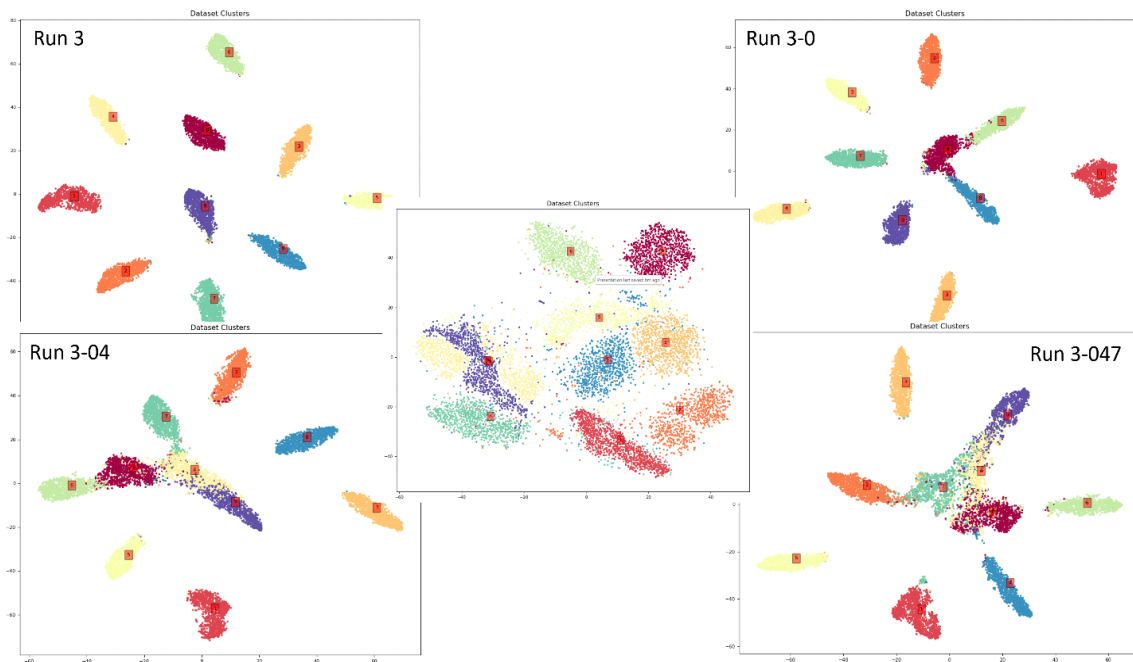


Figure 37: Visualization of Output Clusters

The next area we wanted to investigate a little deeper into was what did the computed output vector look like compared to the one that was determined closest and how did they compare to the targeted value. To answer this question, we generated displays of computed or predicted class, Figure 38,

and its corresponding dictionary value. We also provided the target value and the difference between the predicted and target values. The chart on the right provides the differences as a line chart. The chart provides us with an average distance of 0.21 between the individual predicted and targeted values.

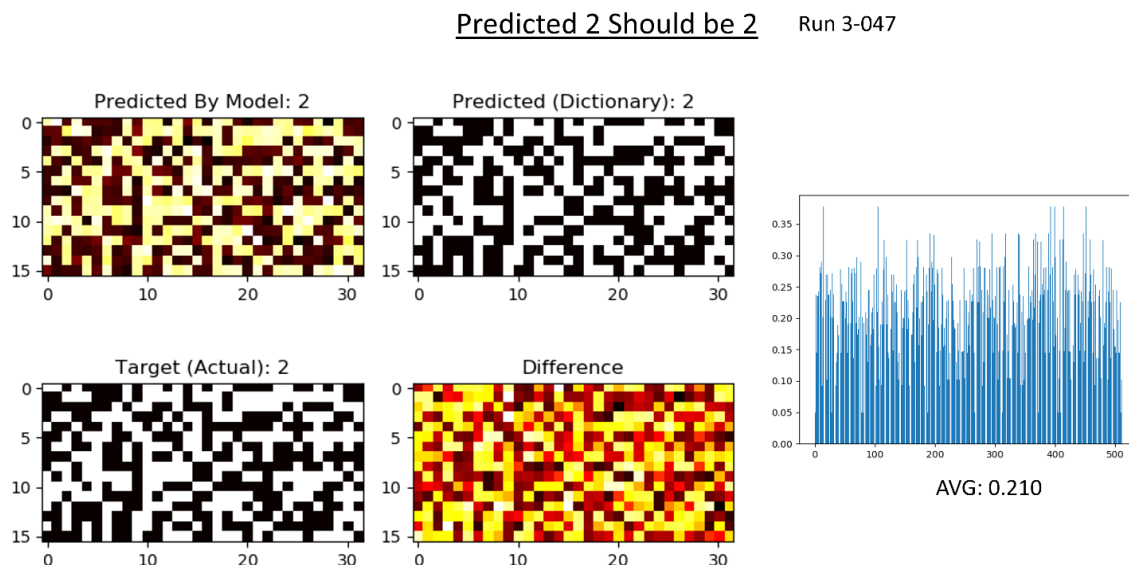


Figure 38: Output Vectors, Correct Prediction

Figure 39 provides an example where the predicted value incorrectly identifies the class. (9 vs 7). We notice in this case a much larger separation between the predicted and targeted values. In this case the average individual difference is 1.02 vs 0.21.

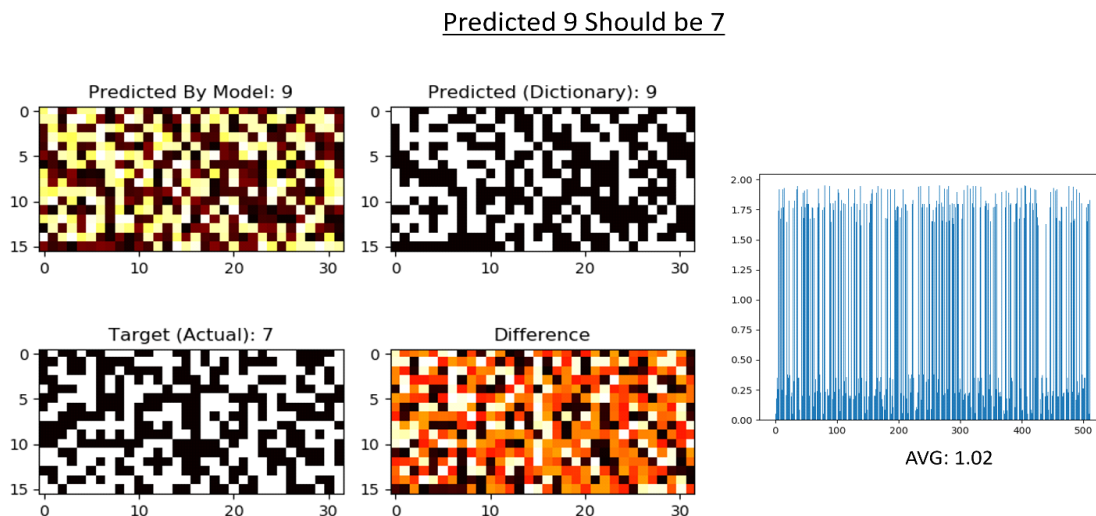


Figure 39: Output Vectors, Wrong Prediction

Besides producing the displays as shown in Figure 38 and Figure 39, we also produced two histograms, Figure 40, that display the distribution of the distances between the predicted and targeted values. Based on what we have produced thus far, we were unable to draw any conclusions as to where a threshold might lie between correct and incorrect predictions.

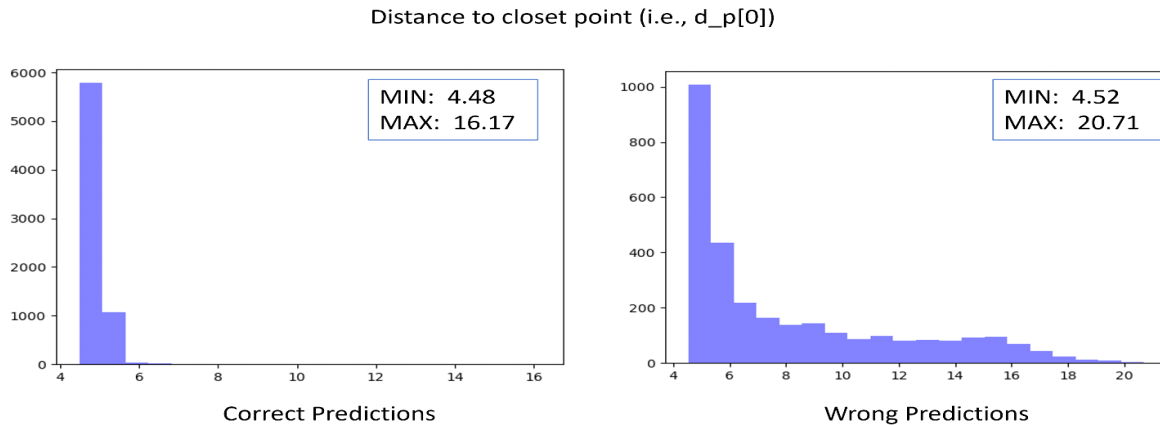


Figure 40: Distance to closet point

4.3.2 Computing Uncertainty.

The next area that we investigated was whether there has been any research determining how well our model is performing. In this case we were interested in looking at how certain or uncertain we are with our predictions. [20-25] is a list of the papers we reviewed. All the papers seemed to talk about uncertainty in theoretical terms. We did not find any papers that attempt to use or implement their findings.

After reading the above documents we came across a YouTube video [21] entitled: “Uncertainty (Aleatoric vs Epistemic) | Machine Learning. While the above references provided a theoretical discussion on uncertainty, none provided any discussion as to how to apply their framework to a real model/application. This video presented a process in which one can develop an uncertainty value. The common theme across the discussions was that the use of dropout during testing could provide sufficient evidence as to the stability/instability in the model’s conclusions. In Figure 41, the model was trained on three classes: sheep, cow, and pig. The model was executed several times (30-50 times, generally referred to a Markovian process) and the result for each class was saved, thus providing a range of answers (as shown in the bottom right triangle). This distribution then defined what the authors called model uncertainty.

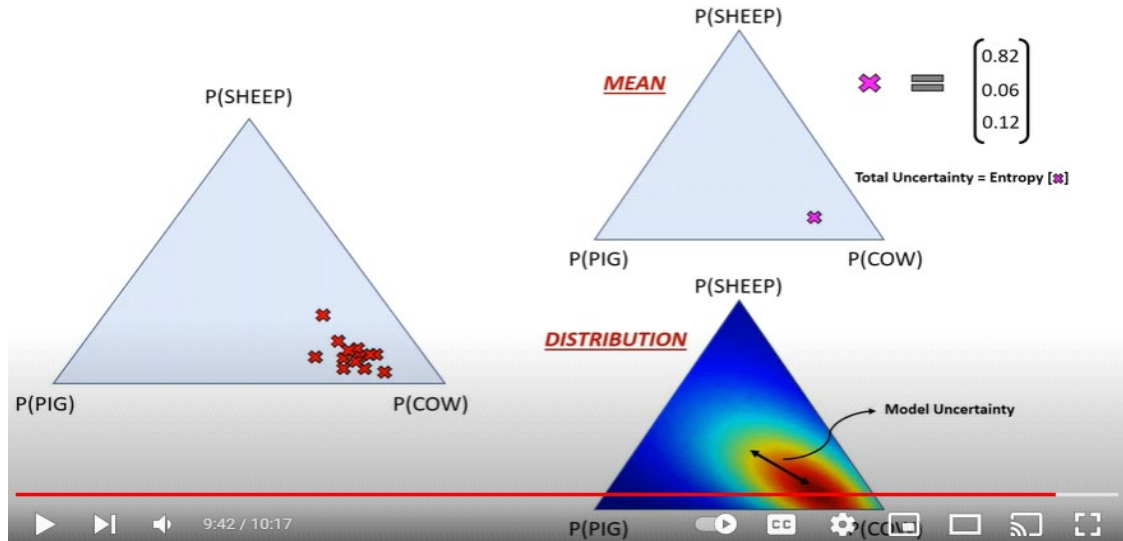


Figure 41: Computing Uncertainty

To compute the model uncertainty, we first had to modify our test program, allowing for multiple runs, and turning dropout on during testing. Typically, dropout is only enabled during the training phase. Before we could do this, we needed to implement dropout in our overall model, but at what rate and at what layer? Table 21 provides several runs implementing dropouts at various layers and at differing rates. We found that for our model adding dropout after each dense or fully connected layers of 0.3 gave us the best overall results.

Table 21: Selecting the Best Locations/Value for Dropout

Run	Validation Set	Tests Set
Convolution Layers Only – DO = 0.5	98.32	98.68
Convolution Layers Only – DO = 0.4	98.68	98.77
Convolution Layers Only – DO = 0.3	98.84	98.92
Convolution Layers Only – DO = 0.2	98.98	98.97
Convolution Layers Only – DO = 0.2	98.94	98.92
No Dropout	99.06	99.30
Dense Layers Only – DO = 0.5	98.76	98.96
Dense Layers Only – DO = 0.4	98.86	98.94
Dense Layers Only – DO = 0.3	99.02	99.12
Dense Layers Only – DO = 0.2	99.21	99.04
Dense Layers Only – DO = 0.1	99.14	99.08

We ran our model for 30 runs and saved the results for each class. Figure 42 provides our results for class 0. The red dots are those results that were misclassified, i.e., those classified as a zero but should have been classified as some other class.

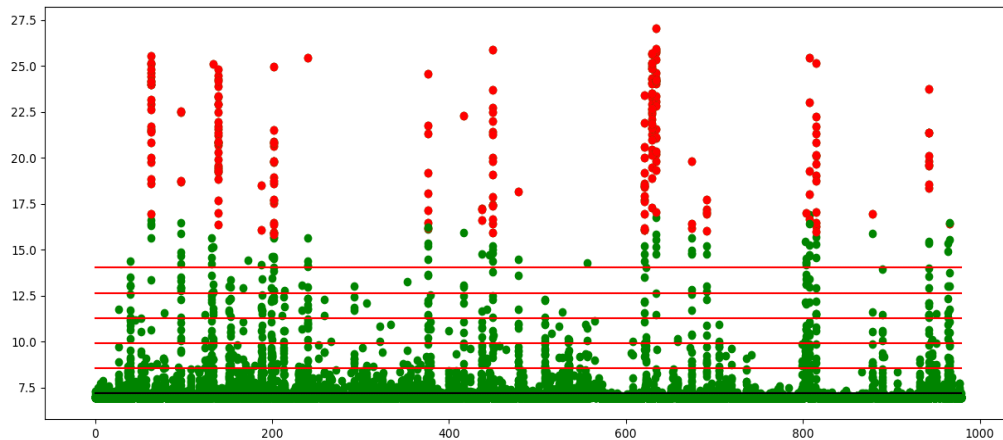


Figure 42: Class 0 Results

For each class, the mean and standard deviation were computed based on the difference between the computed and targeted values. The black line is the overall mean, while the red lines indicate how many sigmas (or standard deviations) the value is from the mean. Below are the computed values.

In all cases (i.e., classes) it was found that 3-sigma works. That is if you were within 3-sigma of the mean you could be certain that the value that the model has provided is correct. As the sigma goes up you will have less certainty in the answer. How much is yet to be proven or if this approach makes sense.

4.3.3 Integrating Prediction and Uncertainty in Our GUI

We added additional visualization tools to our framework. The first tool Figure 43 displays the training and validation accuracies across the complete training scenario. The data is pulled from each of the train.log files. Each run (one for each learning rate) generates such a log. The tool automatically checks/stitches up to three runs together at ‘best’ value achieved from the previous run. It also displays the “best” accuracy for each run.

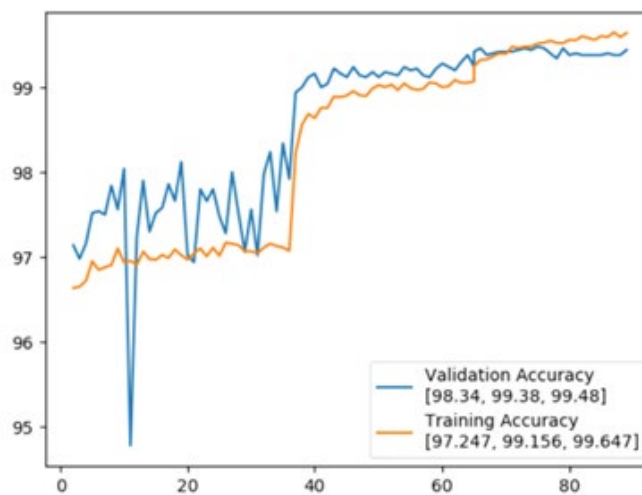


Figure 43: Results Display

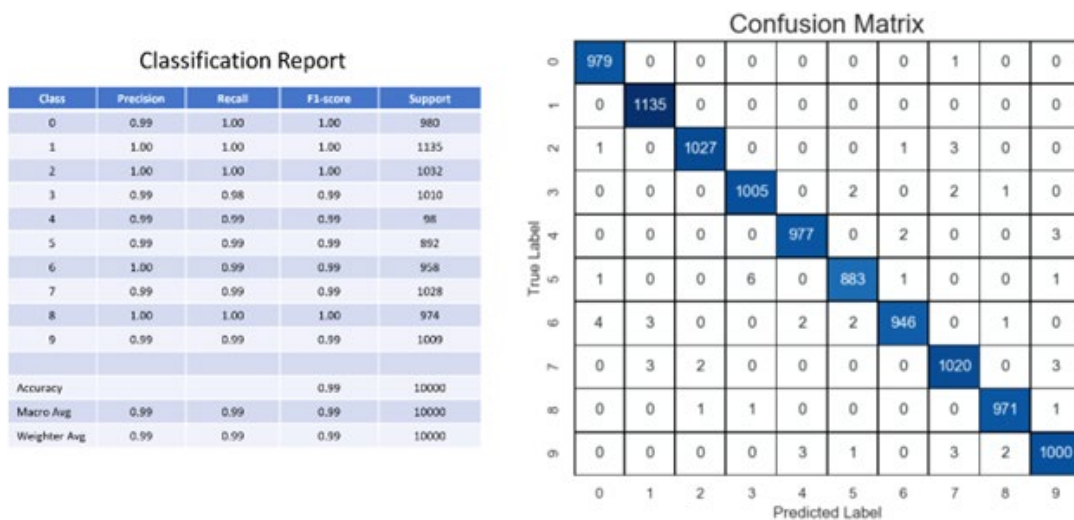


Figure 44: Classification Report/Confusion Matrix

Figure 50 displays the final enhancement we made to our GUI. When entering the image and clicking on the “Display Prediction” button, a confidence value will be displayed in the console window. We have not yet added it to the display. We are still determining where to add it. The final enhancement we made was in the GUI. When entering the image and clicking on the “Display Prediction” button, a confidence value will be displayed in the console window. We have not yet added it to the display. We are still determining where to add it.

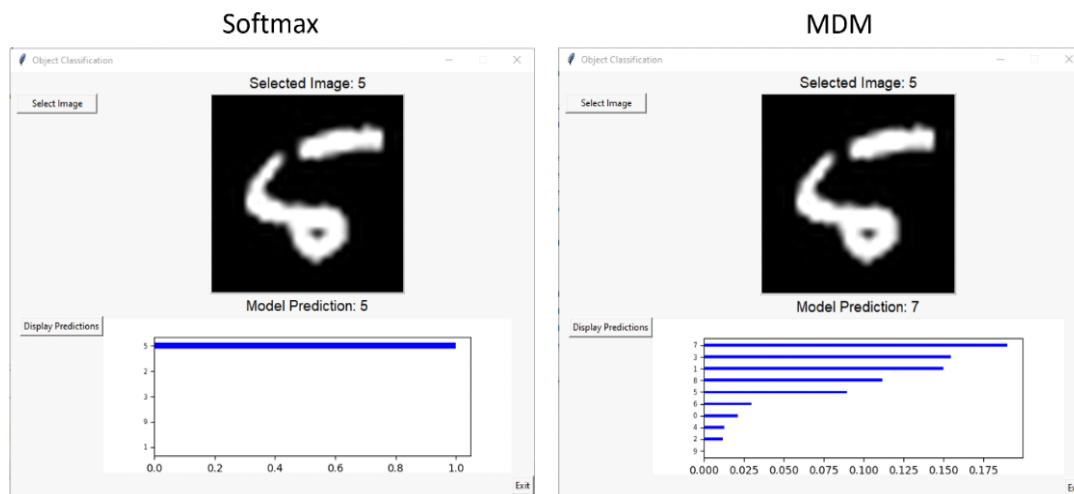


Figure 45: Revised GUI

To compute the uncertainty (refer to Figure 46), we saved the means and standard deviations for each class. We simply started with the distance to the nearest neighbor and subtracted the classes’ mean and divided by the standard deviation. We then took its ceiling.

```

def getUncertaintyMeasure(self, label, dist):

    measures = [[7.2108, 1.3618], [6.6176, 1.6813], [7.3631,
1.3727], [7.2534, 1.7744], [7.6658, 1.7465], [8.0081,
1.6139], [7.4812, 2.3631], [7.3095, 1.9296], [7.5823,
1.7465], [8.0432, 2.6523]]

    mean = measures[label][0]
    std = measures[label][1]
    sigma = math.ceil((dist - mean) / std)

    return sigma

```

Figure 46: Code - Computing Uncertainty

Up until now we have been working with various model architectures to increase their overall performance. As one can see we have a long way to go. Or do we? In the next two sections we look at alternative models. The first is one that starts to address the life-long learning problem while the second one looks at dealing with the real-world, i.e., using labeled and unlabeled data. These two models start to address major problems when it comes to the usability of such models.

4.4 Sparse High-Level-Exclusive, Low-level-Shared (SHELS)

Much of the previous research/successes in deep neural networks has been with models that are trained to learn a fixed set of examples and distributions (generally referred to as the closed world). As we begin to achieve some success with them there is a question or concern for many as to how well they will operate within the real open world where they will be presented with new and never-before-seen patterns. This is a major focus of DARPA’s Lifelong Learning program. An effort under this program, looking at continuous learning is “SHELS: Exclusive Feature Sets for Novelty Detection and Continual Learning Without Class Boundaries“, by Meghna Gummadi, David Kent, Jorge A. Mendez, Eric Eaton from the University of Pennsylvania [25]. After reading the paper and discussing it at our in-house meeting, the team felt it was worth changing our current direction (at the time) and looking into this work in more detail.

4.4.1 SHELS Concept Overview.

An overview of the SHELS process is summarized in Figure 47. A model is trained on a given dataset (e.g., MNIST or CIFAR 10/100). The MNIST database is a large database of handwritten digits that is commonly used for training various image processing systems. The CIFAR-10 dataset (Canadian Institute for Advanced Research) is a collection of images that are commonly used to train machine learning and computer vision algorithms. The baseline models implemented under SHELS are generic but do require some extensions. Besides the last layer being trained using a SoftMax output they also employ cosine normalization in place of the usual dot product after training. The next two parts of their process (that uses a second script) is the Out-Of-Distribution (OOD) Detection and Accommodation phase (or Novelty Detection and Accommodation). When a new pattern is presented to the model it must first determine if the pattern has been seen before (known). If determined to be new or novel it, then needs to learn the new category.

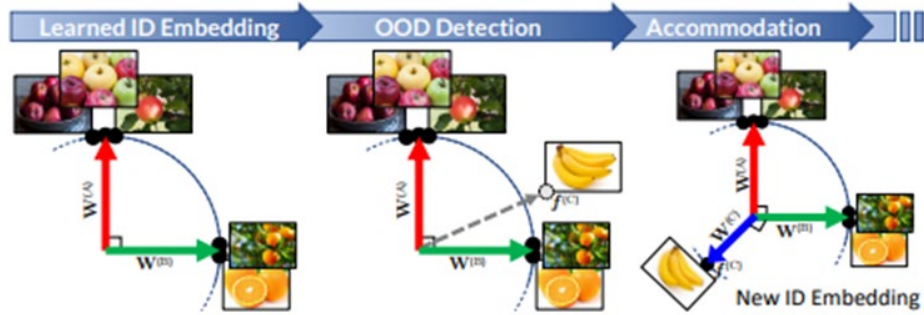


Figure 47: Overall Process (author's paper, page 2 in [25])

In the following sections we discuss three of the major concepts behind SHELS: sparsity, novelty detection and novelty accommodation. Again, here we only provide a brief introduction to these concepts. For more details we refer you to the paper.

4.4.1.1 Sparsity.

The authors of SHELS implemented or experimented with two regularization approaches: exclusivity and group sparsity. Figure 48 provides examples of both. We discuss these two approaches here only to provide context when we conduct our experimental runs in later sections. Exclusivity encourages each upper-level node to choose an exclusive set of lower-level nodes (matching-colored edges). Group Sparsity eliminates neurons that are redundant or not shared by multiple upper-level nodes (dashed-edge neurons). The authors found that group sparsity outperformed exclusivity and based their novelty accommodation (learning) approach on it.

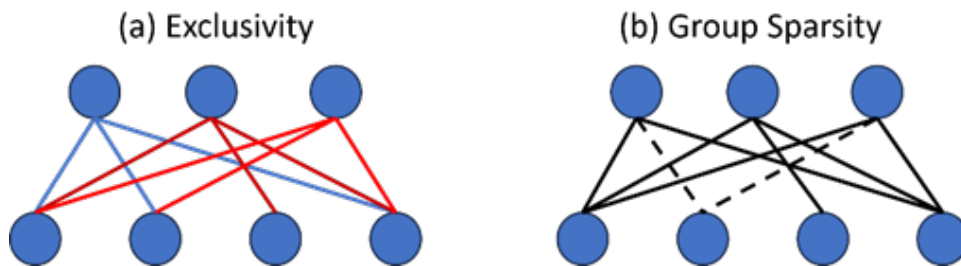


Figure 48: Exclusivity vs Group Sparsity

4.4.1.2 Novelty Detection.

After the model has been trained, it can then be presented with a new pattern. At this point it must be determined whether the new pattern or example is known, is close to a known entity or completely unknown or novel. The way in which SHELS does this is by computing a threshold for each known category. This is accomplished by computing the mean across all patterns for that category minus one standard deviation. Figure 49 provides an overview of the process.

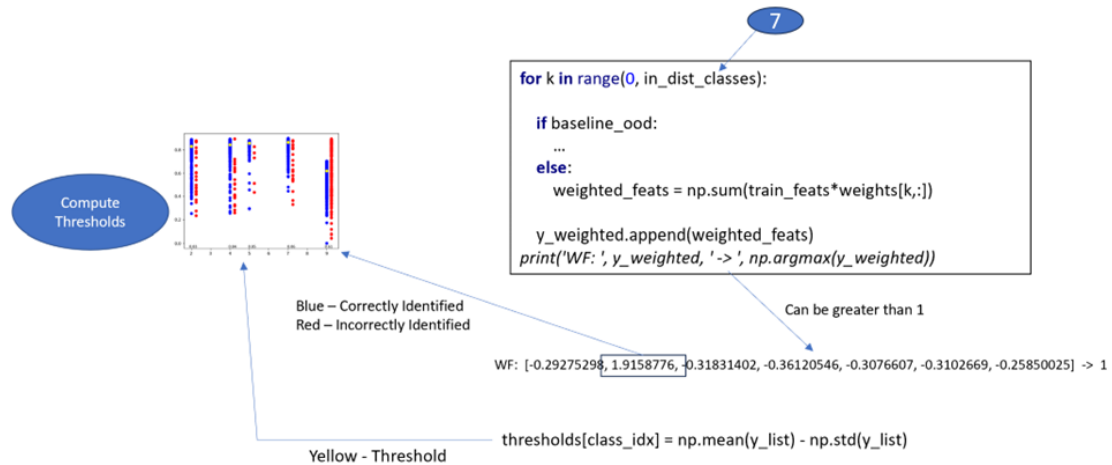


Figure 49: Computing Thresholds

Figure 50 is a chart that we have created to show the predictions (right/wrong for each category) and the computed threshold (yellow). The data is generated for the MNIST dataset, Experiment 0. Five examples were given to the model which we refer to as the In-Distribution (IND) (9, 5, 2, 4, 7) to learn while five additional ones (1, 0, 8, 6, 3) were held out (and referred to as the OOD).

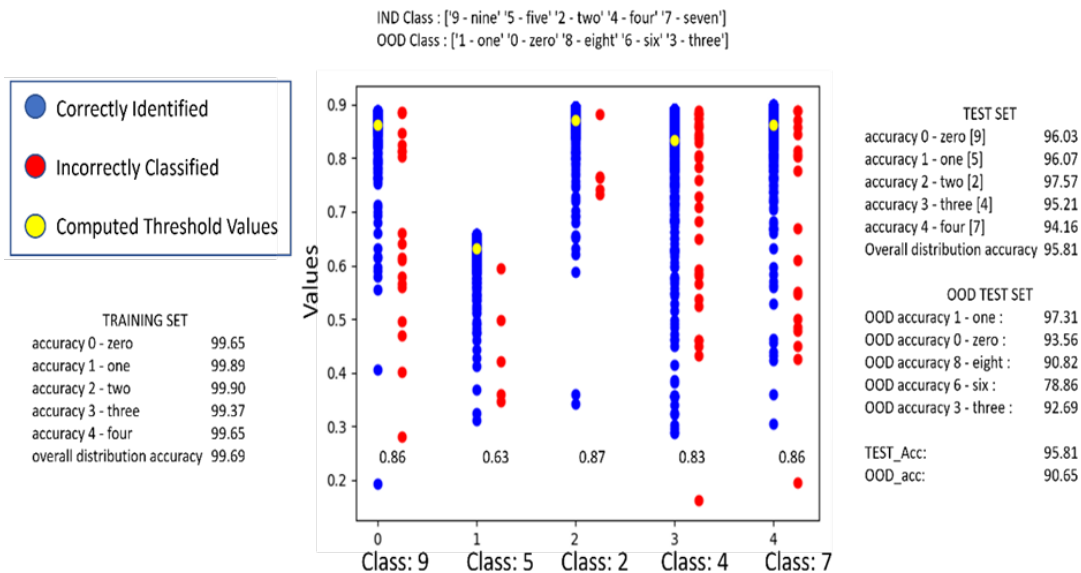


Figure 50: Display of Output Values

At inference time, the authors will use the above thresholds and compute the similarity score of the new pattern with respect to each known class and find the maximum over the classes. Those exceeding a given threshold are considered novel or never seen and take us to the next step, novelty accommodation.

4.4.1.3 Novelty Accommodation.

Once we believe we have a new (novel) category we need to get the model to learn it. However, it has been found by several previous researchers that providing new data to a neural model and incrementally learning it will over time lead to catastrophic forgetting (i.e., forgetting previously learned patterns). The authors' approach focuses on the learning of the new class on unimportant nodes while ensuring that important nodes are not modified. We refer one to Figure 51.

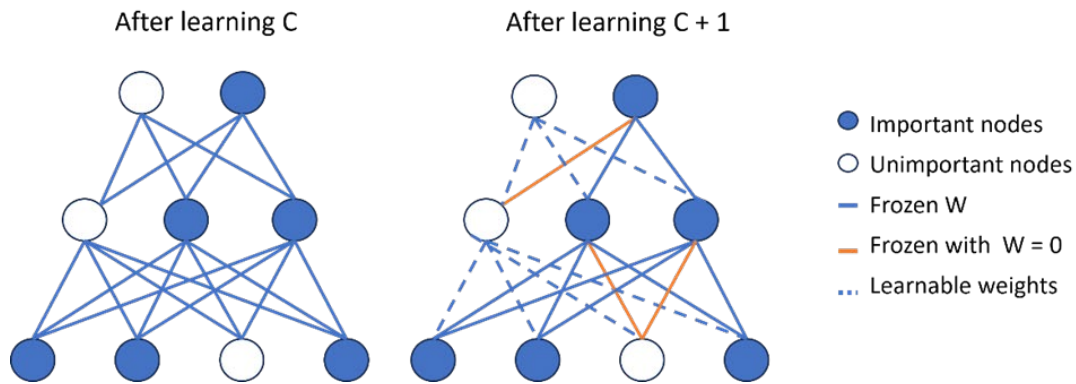


Figure 51: Novelty accommodation via selective weight updates (author's paper, page 5 in [25])

The measure of a node's importance is computed by taking the average activation of each node over the entire training data. They target two types of modifications that could affect important nodes: model drift and negative knowledge transfer. Model drift is defined as when the incoming weights of important nodes are changed during the learning of the new class. To avoid model drift, they penalize the updates. In the second case, negative knowledge transfer occurs when an unimportant node is updated and used as input to an important node. To avoid negative knowledge transfer they set the weights connecting unimportant nodes to zero. This ensures that the new class is primarily learnt in unimportant nodes, without modifying important nodes. The process begins from the bottom up like the training of a neural model. Weights are updated back through the network. For a detailed explanation of the above process, we refer one to Jung, Ahn, Cha, and Moon paper, entitled "Continual learning with node-importance based adaptive group sparse regularization"

Figure 52 provides a flow chart with a little more detail as to the learning process. The code first determines whether it is a new pattern (currently hardcoded) and if so, enters the "Incremental Learning" portion of the code. At this point they learn only the new pattern adjusting the weights as described above. Once completed they then send the entire model to the "Continuous Learning" phase. At this point they adjust the weights for a fixed number (13) of epochs. Once completed the previous thresholds are updated and checked to see if there is another pattern to be learned. If not, it then performs the continuous learning portion one final time.

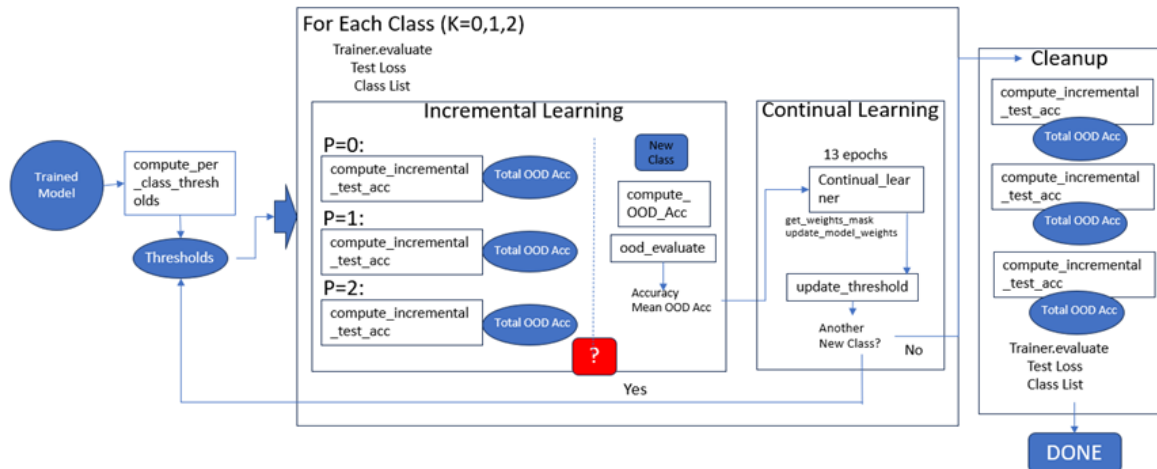


Figure 52: Novelty Detection

Figure 53 displays the results after running the detection and accommodation script once the training of the model was completed. The orange line depicts the scenario. The authors present the first new pattern to the model and compute the overall accuracy. As expected, the overall accuracy goes down. They next ran their accommodation process (where a new category is detected and added). We see after this the overall accuracy increases (but not back to the level they started with). They repeat this for the other two examples. One can see that over time the overall accuracy does decrease as new categories are added but overall, the results are very promising.

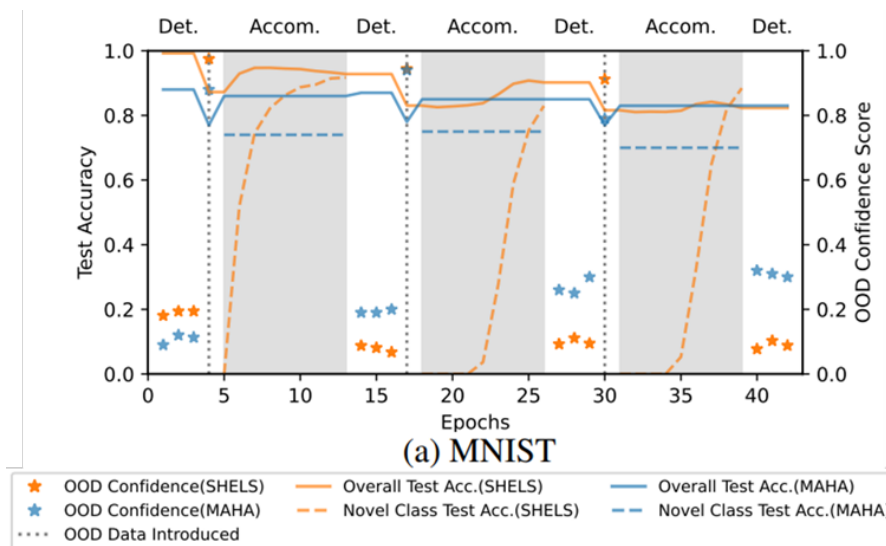


Figure 53: Results of Incremental Learning (author's paper, page 10 in [25])

Our goal was to reproduce the chart provided in Figure 7 of the paper. We could not. After a conversation with the authors, we found that the scripts provided under GIT were missing a couple of arguments/options. These included: “--full_pipeline True and --cont_learner True.” Once we set these values, we were able to get the data required to reproduce the chart. Appendix A (second part) provides the log of running our first experiment learning three new classes.

4.4.2 Modifications Required.

In this section we discuss the changes that we made to the baseline code to get it to run under Windows 11 and Pycharm. The code was available via github; however, it was written for a Linux operating system, requiring additional parameter configuration to run on Windows. For example, one must separately download/install a version of torch that uses the CUDA libraries (GPU) for parallelization. It is not installed by default. We installed (using pip) version 1.13.1 of torch which is supported by CUDA libraries version 11.6. The actual command to install this version along with other necessary libraries is shown below:

```
pip install torch==1.13.1+cu116 torchvision==0.14.1+cu116 torchaudio==0.13.1 --extra-index-url  
https://download.pytorch.org/whl/cu116
```

After we installed torch and any of the other required libraries, we also had to make some changes to the baseline code. Below is a summary of the changes.

- To get train (Section 3) to run
 - In the model_cosine1 we had to comment out line 41 and bn() removed from line 61
 - This was due to if the batch size = 1 it produces an error in computing the batch normalization.
 - After discussions with one of the authors (Meghna) told me they did not use it for their model – only their baseline model; suggested leaving it commented out.
 - It does, however, change the baseline results, in fact 8/10 did not train. Left in during training of baseline runs. Not sure what to do yet for novelty/accommodation.
- To get accommodation/detection to work we needed the following changes:
 - Uninstall pytorch and install pytorch for cuda.
 - layer_utils.py
 - Line 39: avg_layers.append(mean_layer.cpu().numpy()).
 - Line 41: avg_act_all_layers+=np.array(avg_layers, dtype=object).
 - weight_freezing.py
 - Inserted line in front of line 30: activations = torch.from_numpy(activations).float().
 - Line 62: activations = layer_act[layer_idx[curr_idx]] – removed .cpu().numpy() Line 83: activations = layer_act[layer_idx[curr_idx]] – removed .cpu().numpy().

We were able to run all the sample scripts after making the above changes.

4.4.3 Experimental Results.

In this section we provide the results of our experiments using the SHELS code. Up to this point (Train (Sec 3)) we were successful in demonstrating that the initial framework can support various models. The basic models (SHELS and Baseline) were provided as part of the baseline. They also provided a VGG-16 model. We were able to successfully add LeNet-5, AlexNet, VGG-19 and ResNet18. The only concern with Resnet18 is that each layer within the model is not individually addressable or hierarchical as it is with all the other models. The ResNet model also has feedback. The model ran but we are unsure as to how well any of them did in the detection and accommodation phases. Figure 54 summarizes what tasks we have accomplished at this point.

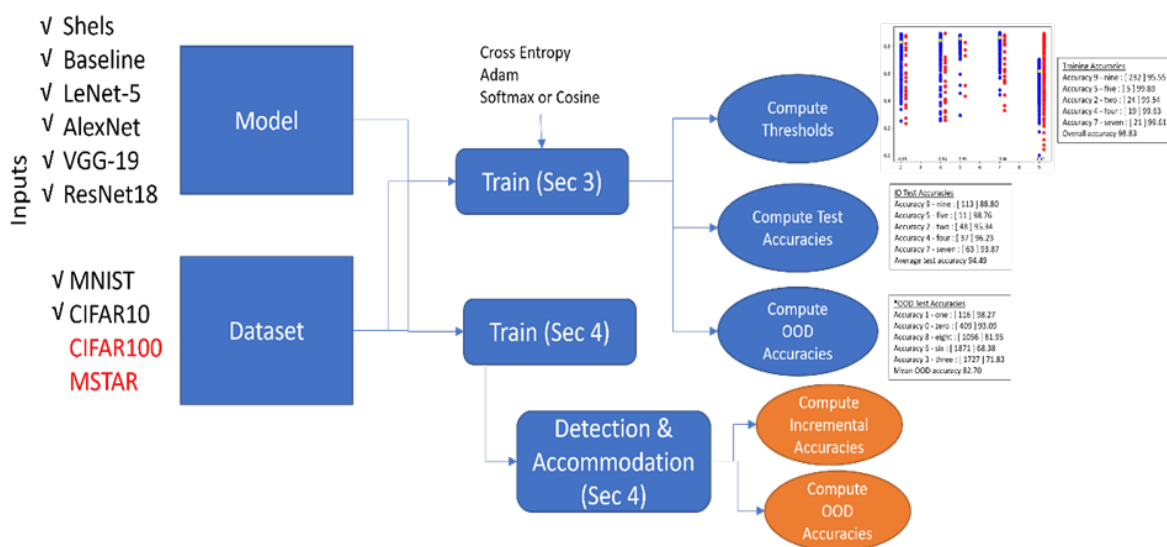


Figure 54: High-Level Perspective of Work Tasks

The second half of the SHELS readme described 3 additional scripts (Sec 4 in Figure 4.4.8): (1) Train, (2) Accommodation and (3) Accommodation and Detection. Since we were interested in the incremental learning portion of the model, we did not exercise the accommodation only script (and do not show it in Figure 4.4.8). Unlike the runs we conducted under the first set of scripts, the number of categories were increased to 7 (versus 5). These runs set the baselines from which the other two scripts start with. We ran all our models through the Training script. Table 22 provides the results and a comparison of the effect that the learning rate has on the overall accuracy of the model. The black numbers were computed using the same learning rate of 0.01 for each model. The red values demonstrate the effect of reducing the learning rate. As the model gets larger (number of layers/parameters), lower learning rates seem to work much better.

Table 22: Results of Adding New Model Architectures

EXP		MNIST	CIFAR10				
	Classes		Train/Test Accuracy (%)				
		model_cosine1 LR=0.0001	LeNet-5 LR =0.001	model_cosine_cifar LR=0.0001	AlexNet LR=0.0001	VGG-19	Resnet18 LR=0.0001
0	ID [9, 5, 2, 4, 7, 1, 0] OOD [8, 6, 3]	99.29/99.31 99.65/99.39	73.84/80.14 70.21/70.20	77.72/82.50 76.56/81.33	73.23/75.17 86.51/87.62	59.78/59.82	93.80/95.76 93.80/95.76
1	ID [2, 1, 0, 4, 6, 5, 7] OOD [8, 3, 9]	99.49/99.45 99.73/99.43	75.55/82.88 70.79/66.95	79.54/83.57 77.72/82.84	85.55/89.73 87.00/85.05	74.71/75.17	93.96/98.12 93.96/98.12
2	ID [2, 4, 8, 3, 9, 0, 5] OOD [7, 6, 1]	99.22/99.05 99.72/99.56	64.10/68.42 61.32/65.27	67.74/73.27 67.25/71.89	61.83/64.58 73.32/73.95	49.95/41.95	80.89/88.42 80.89/88.42
3	ID [6, 9, 3, 5, 7, 0, 2] OOD [8, 4, 1]	99.25/99.14 99.70/99.78	69.80/76.30 63.92/60.05	73.08/77.60 70.50/75.48	72.33/72.89 73.49/70.62	63.75/65.77	92.25/94.77 92.25/94.77
4	ID [6, 0, 2, 3, 9, 1, 3] OOD [4, 5, 7]	99.46/99.49 99.79/99.49	74.42/79.88 72.44/76.53	80.05/84.45 76.98/78.52	73.10/76.78 85.98/66.97	50.81/52.11	94.12/96.44 94.12/96.44
5	ID [2, 0, 6, 3, 8, 7, 5] OOD [9, 4, 1]	99.31/98.79 99.75/99.56	69.94/74.67 62.34/65.83	71.05/72.06 69.13/70.83	72.73/76.61 73.96/75.94	53.08/46.54	90.63/94.57 90.63/94.57
6	ID [5, 2, 6, 7, 8, 0, 1] OOD [4, 3, 9]	99.31/98.97 99.70/99.58	77.72/84.09 73.79/76.14	81.35/86.20 79.43/83.64	86.92/89.63 87.01/87.28	76.96/74.43	94.61/96.42 94.61/96.42
7	ID [0, 8, 3, 6, 7, 1, 9] OOD [4, 5, 2]	99.35/99.32 99.77/99.65	76.30/82.22 74.65/64.84	81.77/85.8 80.39/82.61	87.55/92.64 88.09/85.29	54.60/50.80	95.26/97.89 95.26/97.89
8	ID [2, 6, 3, 8, 1, 7, 0] OOD [4, 5, 9]	99.42/99.42 99.76/99.55	76.47/83.03 72.92/71.37	79.20/84.92 78.00/82.33	85.70/84.90 86.89/86.89	73.03/73.27	94.30/97.51 94.30/97.51
9	ID [6, 9, 7, 1, 5, 8, 0] OOD [4, 2, 3]	99.33/99.21 99.74/99.45	76.37/82.41 76.46/79.52	81.35/86.20 81.76/87.21	86.92/89.63 89.12/75.30	56.18/56.56	95.34/94.63 95.34/94.63
AVERAGES		99.34/99.22 99.73/99.54	73.45/79.40 69.88/69.67	77.29/81.66 75.77/79.67	78.59/81.45 83.14/74.49	61.29/59.64	92.52/95.45 92.52/95.45

NOTE: Accuracies different, trying to learn 7 vs 5 classes with same number of epochs (10), except for ResNet model?

With last BN commented out – fails if only have one training pattern in last set, cannot compute a mean with only one value

At this stage we found that the code had difficulty running many of our models except for the two that came with the system. These included the MNIST and CIFAR10 models. So, we limited our experiments at this point to only these two models and for experiment 0. The original baseline runs ran 10 experiments (0 – 9) but the example used for the detection and accommodation phase only used the results from experiment 0. To save time we modified the code to train only on experiment 0.

Table 23, Table 24 and Table 25 provide the results for training our MNIST model across three different learning rates (0.001, 0.0001, and 0.00001) for 10, 25 and 50 epochs. We also looked at whether exclusivity and/or group sparsity to true or false mattered. The red colored values indicate the best validation rates for each epoch set. Our overall best results provided a validation accuracy of 99.72 %. This was achieved in two cases: when exclusivity and group sparsity were set to False and learning rate = 0.001 at 33 epochs and learning rate of 0.0001 at epoch 26.

Table 23: MNIST (Learning Rate = 0.001)

Directory	Exclusivity	Group Sparsity	Learning rate	Accuracies for Exp 0 (Best Run of 10)	Accuracies for Exp 0 (Best Run of 25)	Accuracies for Exp 0 (Best Run of 50)
dir_MNIST_train_ft00110/25/50	DEFAULT (False)	True	0.001	EPOCH: 10 time elapsed: 31.238 Train Loss: 1.239 Train Acc: 99.57% Test Loss: 1.241 Test Acc: 99.38% Valid Loss: 1.240 Valid Acc: 99.29%	EPOCH: 20 time elapsed: 30.326 Train Loss: 1.336 Train Acc: 99.73% Test Loss: 1.337 Test Acc: 99.42% Valid Loss: 1.338 Valid Acc: 99.45%	EPOCH: 38 time elapsed: 30.717 Train Loss: 1.343 Train Acc: 99.89% Test Loss: 1.346 Test Acc: 99.51% Valid Loss: 1.347 Valid Acc: 99.53%
dir_MNIST_train_tt00110/25/50	True	True	0.001	EPOCH: 4 time elapsed: 30.534 Train Loss: 1.141 Train Acc: 98.78% Test Loss: 1.141 Test Acc: 98.81% Valid Loss: 1.143 Valid Acc: 98.46%	EPOCH: 12 time elapsed: 32.127 Train Loss: 1.220 Train Acc: 98.80% Test Loss: 1.220 Test Acc: 98.81% Valid Loss: 1.220 Valid Acc: 98.62%	EPOCH: 47 time elapsed: 43.777 Train Loss: 1.249 Train Acc: 98.63% Test Loss: 1.249 Test Acc: 98.78% Valid Loss: 1.247 Valid Acc: 98.69%
dir_MNIST_train_tf00110/25/50	True	DEFAULT (False)	0.001	EPOCH: 4 time elapsed: 28.112 Train Loss: 1.123 Train Acc: 99.09% Test Loss: 1.123 Test Acc: 99.12% Valid Loss: 1.125 Valid Acc: 98.93%	EPOCH: 4 time elapsed: 27.966 Train Loss: 1.123 Train Acc: 99.09% Test Loss: 1.123 Test Acc: 99.12% Valid Loss: 1.125 Valid Acc: 98.93%	EPOCH: 4 time elapsed: 29.218 Train Loss: 1.123 Train Acc: 99.09% Test Loss: 1.123 Test Acc: 99.12% Valid Loss: 1.125 Valid Acc: 98.93%
dir_MNIST_train_ff00110/25/50	DEFAULT (False)	DEFAULT (False)	0.001	EPCH: 7 time elapsed: 27.933 Train Loss: 1.115 Train Acc: 99.69% Test Loss: 1.116 Test Acc: 99.51% Valid Loss: 1.117 Valid Acc: 99.41%	EPOCH: 24 time elapsed: 26.821 Train Loss: 1.112 Train Acc: 99.89% Test Loss: 1.116 Test Acc: 99.48% Valid Loss: 1.114 Valid Acc: 99.66%	EPOCH: 33 time elapsed: 28.072 Train Loss: 1.111 Train Acc: 99.95% Test Loss: 1.115 Test Acc: 99.59% Valid Loss: 1.114 Valid Acc: 99.72%

Table 24: MNIST (Learning Rate = 0.0001)

Directory	Exclusivity	Group Sparsity	Learning rate	Accuracies for Exp 0 (Best Run of 10)	Accuracies for Exp 0 (Best Run of 25)	Accuracies for Exp 0 (Best Run of 50)
dir_MNIST_train_ft000110/25/50	DEFAULT (False)	True	0.0001	EPOCH: 8 time elapsed: 34.247 Train Loss: 1.116 Train Acc: 99.74% Test Loss: 1.118 Test Acc: 99.59% Valid Loss: 1.119 Valid Acc: 99.41%	EPOCH: 22 time elapsed: 29.947 Train Loss: 1.126 Train Acc: 99.87% Test Loss: 1.128 Test Acc: 99.65% Valid Loss: 1.129 Valid Acc: 99.68%	EPOCH: 22 time elapsed: 28.804 Train Loss: 1.126 Train Acc: 99.87% Test Loss: 1.128 Test Acc: 99.65% Valid Loss: 1.129 Valid Acc: 99.68%
dir_MNIST_train_tt000110/25/50	True	True	0.0001	EPOCH: 10 time elapsed: 31.600 Train Loss: 1.121 Train Acc: 99.41% Test Loss: 1.121 Test Acc: 99.36% Valid Loss: 1.123 Valid Acc: 98.99%	EPOCH: 19 time elapsed: 31.099 Train Loss: 1.123 Train Acc: 99.46% Test Loss: 1.124 Test Acc: 99.35% Valid Loss: 1.126 Valid Acc: 99.23%	EPOCH: 37 time elapsed: 32.519 Train Loss: 1.134 Train Acc: 99.46% Test Loss: 1.135 Test Acc: 99.35% Valid Loss: 1.136 Valid Acc: 99.25%
dir_MNIST_train_tf000110/25/50	True	DEFAULT (False)	0.0001	EPOCH: 10 time elapsed: 28.403 Train Loss: 1.120 Train Acc: 99.27% Test Loss: 1.120 Test Acc: 99.24% Valid Loss: 1.122 Valid Acc: 99.13%	EPOCH: 15 time elapsed: 33.654 Train Loss: 1.117 Train Acc: 99.50% Test Loss: 1.118 Test Acc: 99.42% Valid Loss: 1.119 Valid Acc: 99.27%	EPOCH: 49 time elapsed: 28.492 Train Loss: 1.117 Train Acc: 99.50% Test Loss: 1.118 Test Acc: 99.42% Valid Loss: 1.118 Valid Acc: 99.45%
dir_MNIST_train_ff000110/25/50	DEFAULT (False)	DEFAULT (False)	0.0001	EPOCH: 9 time elapsed: 25.559 Train Loss: 1.113 Train Acc: 99.85% Test Loss: 1.115 Test Acc: 99.65% Valid Loss: 1.116 Valid Acc: 99.62%	EPOCH: 9 time elapsed: 26.829 Train Loss: 1.113 Train Acc: 99.85% Test Loss: 1.115 Test Acc: 99.65% Valid Loss: 1.116 Valid Acc: 99.62%	EPOCH: 26 time elapsed: 26.251 Train Loss: 1.111 Train Acc: 99.96% Test Loss: 1.113 Test Acc: 99.77% Valid Loss: 1.115 Valid Acc: 99.72%

Table 25: MNIST (Learning Rate = 0.00001)

Directory	Exclusivity	Group Sparsity	Learning rate	Accuracies for Exp 0 (Best Run of 10)	Accuracies for Exp 0 (Best Run of 25)	Accuracies for Exp 0 (Best Run of 50)
dir_MNIST_train_ft0000110/25/50	DEFAULT (False)	True	0.00001	EPOCH: 10 time elapsed: 32.437 Train Loss: 1.198 Train Acc: 99.08% Test Loss: 1.198 Test Acc: 99.10% Valid Loss: 1.200 Valid Acc: 98.79%	EPOCH: 25 time elapsed: 30.388 Train Loss: 1.128 Train Acc: 99.53% Test Loss: 1.130 Test Acc: 99.38% Valid Loss: 1.131 Valid Acc: 99.25%	EPOCH: 43 time elapsed: 30.329 Train Loss: 1.115 Train Acc: 99.82% Test Loss: 1.118 Test Acc: 99.49% Valid Loss: 1.119 Valid Acc: 99.47%
dir_MNIST_train_tt0000110/25/50	True	True	0.00001	EPOCH: 10 time elapsed: 34.070 Train Loss: 1.191 Train Acc: 99.04% Test Loss: 1.191 Test Acc: 99.07% Valid Loss: 1.193 Valid Acc: 98.75%	EPOCH: 24 time elapsed: 32.434 Train Loss: 1.126 Train Acc: 99.53% Test Loss: 1.126 Test Acc: 99.48% Valid Loss: 1.129 Valid Acc: 99.19%	EPOCH: 49 time elapsed: 32.383 Train Loss: 1.117 Train Acc: 99.71% Test Loss: 1.119 Test Acc: 99.59% Valid Loss: 1.121 Valid Acc: 99.41%
dir_MNIST_train_tf0000110/25/50	True	DEFAULT (False)	0.00001	EPOCH: 10 time elapsed: 31.867 Train Loss: 1.187 Train Acc: 99.08% Test Loss: 1.187 Test Acc: 99.14% Valid Loss: 1.189 Valid Acc: 98.75%	EPOCH: 24 time elapsed: 31.133 Train Loss: 1.122 Train Acc: 99.57% Test Loss: 1.123 Test Acc: 99.56% Valid Loss: 1.126 Valid Acc: 99.23%	EPOCH: 49 time elapsed: 30.711 Train Loss: 1.116 Train Acc: 99.76% Test Loss: 1.118 Test Acc: 99.52% Valid Loss: 1.119 Valid Acc: 99.41%
dir_MNIST_train_ff0000110/25/50	DEFAULT (False)	DEFAULT (False)	0.00001	EPOCH: 10 time elapsed: 42.868 Train Loss: 1.191 Train Acc: 99.12% Test Loss: 1.191 Test Acc: 99.10% Valid Loss: 1.193 Valid Acc: 98.75%	EPOCH: 24 time elapsed: 27.408 Train Loss: 1.123 Train Acc: 99.65% Test Loss: 1.124 Test Acc: 99.48% Valid Loss: 1.126 Valid Acc: 99.31%	EPOCH: 43 time elapsed: 27.410 Train Loss: 1.114 Train Acc: 99.86% Test Loss: 1.117 Test Acc: 99.53% Valid Loss: 1.118 Valid Acc: 99.49%

The next step was to run the detection and accommodation algorithm on the new patterns. Table 26 and Table 27 provide the results of this phase. We used our model that was trained using the learning rate = 0.0001. We investigated the impact of both sparsity options and number of training epochs (as part of the continual learning portion) used as part of this phase. The best results were running our trained model with exclusivity = false, group sparsity = true and setting the options for the

detection and accommodation phase the same. The values shown in each column entry are the results of running the detection and accommodation code (refer back to Figure 60).

Table 26: MNIST Detection & Accommodation Results

Directory	Exclusivity	Group Sparsity	Learning rate	Accuracies for Exp 0 (Best Run of 13)		Accuracies for Exp 0 (Best Run of 13), 25 Epochs	
dir_MNIST_train_ft000110 False, True	DEFAULT (False)	True	0.0001	Acc with 7 classes [9 5 2 4 7 1 0]:	99.59%	Acc with 7 classes [9 5 2 4 7 1 0]:	99.65%
				Accuracy 8 - eight : [3]	99.49%	Accuracy 8 - eight : [8]	98.97%
				Acc with 8 classes [9 5 2 4 7 1 0 8]:	92.41%	Acc with 8 classes [9 5 2 4 7 1 0 8]:	89.67%
				Accuracy 6 - six : [7]	98.82%	Accuracy 6 - six : [17]	97.12%
				Acc with 9 classes [9 5 2 4 7 1 0 8 6]:	82.82%	Acc with 9 classes [9 5 2 4 7 1 0 8 6]:	87.85%
				Accuracy 3 - three : [28]	95.43%	Accuracy 3 - three : [21]	96.57%
				Acc with all 10 classes [9 5 2 4 7 1 0 8 6 3]:	77.94%	Acc with all 10 classes [9 5 2 4 7 1 0 8 6 3]:	80.17%
dir_MNIST_train_tt000110 True, True	DEFAULT (False)	True	0.0001	Acc with 7 classes [9 5 2 4 7 1 0]:	99.36%		
				Accuracy 8 - eight : [13]	98.12%		
				Acc with 8 classes [9 5 2 4 7 1 0 8]:	24.65%		
				Accuracy 6 - six : [440]	24.65%		
				Acc with 9 classes [9 5 2 4 7 1 0 8 6]:	24.65%		
				Accuracy 3 - three : [518]	15.50%		
				Acc with all 10 classes [9 5 2 4 7 1 0 8 6 3]:	24.65%		
dir_MNIST_train_tf000110 True, False	DEFAULT (False)	True	0.0001	Acc with 7 classes [9 5 2 4 7 1 0]:	99.23%		
				Accuracy 8 - eight : [11]	97.78%		
				Acc with 8 classes [9 5 2 4 7 1 0 8]:	12.13%		
				Accuracy 6 - six : [591]	0%		
				Acc with 9 classes [9 5 2 4 7 1 0 8 6]:	12.13%		
				Accuracy 3 - three : [613]	0%		
				Acc with all 10 classes [9 5 2 4 7 1 0 8 6 3]:	12.13%		
dir_MNIST_train_ff000110 False, False	DEFAULT (False)	True	0.0001	Acc with 7 classes [9 5 2 4 7 1 0]:	99.65%		
				Accuracy 8 - eight : [3]	99.49%		
				Acc with 8 classes [9 5 2 4 7 1 0 8]:	12.13%		
				Accuracy 6 - six : [591]	0%		
				Acc with 9 classes [9 5 2 4 7 1 0 8 6]:	12.13%		
				Accuracy 3 - three : [613]	0%		
				Acc with all 10 classes [9 5 2 4 7 1 0 8 6 3]:	12.13%		

Table 27: MNIST Detection & Accommodation Results (Cont'd)

Directory	Exclusivity	Group Sparsity	Learning rate	Accuracies for Exp 0 (Best Run of 13)	
dir_MNIST_train_ft000110 False, True	True	True	0.0001	Acc with 7 classes [9 5 2 4 7 1 0]:	99.59%
				Accuracy 8 - eight : [3]	99.49%
				Acc with 8 classes [9 5 2 4 7 1 0 8]:	89.77%
				Accuracy 6 - six : [7]	98.82%
				Acc with 9 classes [9 5 2 4 7 1 0 8 6]:	82.82%
				Accuracy 3 - three : [28]	95.43%
				Acc with all 10 classes [9 5 2 4 7 1 0 8 6 3]:	73.38%
dir_MNIST_train_ft000110 False, True	True	Default (False)	0.0001	Acc with 7 classes [9 5 2 4 7 1 0]:	99.59%
				Accuracy 8 - eight : [3]	99.49%
				Acc with 8 classes [9 5 2 4 7 1 0 8]:	89.78%
				Accuracy 6 - six : [7]	98.82%
				Acc with 9 classes [9 5 2 4 7 1 0 8 6]:	82.81%
				Accuracy 3 - three : [28]	95.43%
				Acc with all 10 classes [9 5 2 4 7 1 0 8 6 3]:	73.38%
dir_MNIST_train_ft000110 False, True	DEFAULT (False)	Default (False)	0.0001	Acc with 7 classes [9 5 2 4 7 1 0]:	99.59%
				Accuracy 8 - eight : [3]	99.49%
				Acc with 8 classes [9 5 2 4 7 1 0 8]:	12.13%
				Accuracy 6 - six : [591]	0%
				Acc with 9 classes [9 5 2 4 7 1 0 8 6]:	12.13%
				Accuracy 3 - three : [613]	0%
				Acc with all 10 classes [9 5 2 4 7 1 0 8 6 3]:	12.13%

We first ran the test code on all the known categories/data [9, 5, 3, 4, 7, 1, 0] to get an overall accuracy. We next presented the first novel pattern [3]. We computed the accuracy of just the newly learned pattern. We then added the new category and recomputed (for 13 epochs) the overall accuracy. A similar process was run for the other two new classes [3, 8]. We also looked at the effect of increasing the number of epochs (from 13 to 25) during the continual learning process.

This resulted in an overall accuracy of 77.94% for 13 epochs and 80.17% after 25 epochs. Several of the other options, as shown, did not run (indicated in black).

We performed a similar analysis with the CIFAR10 dataset and model. Table 28, Table 29 and Table 30 provide the results for learning rates of 0.01, 0.0001 and 0.00001. Again, we display the best validation accuracies for each set of epochs in red. The overall best results (validation accuracy = 98.14%) were obtained with a learning rate of 0.001.

Table 28: CIFAR10 (Learning Rate = 0.001)

Directory	Exclusivity	Group Sparsity	Learning rate	Accuracies for Exp 0 (Best Run of 10)	Accuracies for Exp 0 (Best Run of 25)	Accuracies for Exp 0 (Best Run of 50)
dir_CIFAR10_train_ft0011 0/25/50	DEFAULT (False)	True	0.001	EPOCH: 10 time elapsed: 29.788 Train Loss: 1.332 Train Acc: 77.61% Test Loss: 1.358 Test Acc: 74.41% Valid Loss: 1.331 Valid Acc: 77.83%	EPOCH: 25 time elapsed: 30.055 Train Loss: 1.357 Train Acc: 75.06% Test Loss: 1.377 Test Acc: 72.65% Valid Loss: 1.355 Valid Acc: 75.17%	EPOCH: 40 time elapsed: 28.678 Train Loss: 1.337 Train Acc: 77.42% Test Loss: 1.358 Test Acc: 74.84% Valid Loss: 1.335 Valid Acc: 78.01%
dir_CIFAR10_train_tt0011 0/25/50	True	True	0.001	EPOCH: 10 time elapsed: 31.058 Train Loss: 1.447 Train Acc: 62.86% Test Loss: 1.452 Test Acc: 61.75% Valid Loss: 1.441 Valid Acc: 63.31%	EPOCH: 25 time elapsed: 30.699 Train Loss: 1.401 Train Acc: 69.94% Test Loss: 1.414 Test Acc: 68.36% Valid Loss: 1.402 Valid Acc: 69.61%	EPOCH: 44 time elapsed: 30.938 Train Loss: 1.367 Train Acc: 73.56% Test Loss: 1.381 Test Acc: 71.53% Valid Loss: 1.368 Valid Acc: 73.12%
dir_CIFAR10_train_tf0011 0/25/50	True	DEFAULT (False)	0.001	EPOCH: 10 time elapsed: 27.910 Train Loss: 1.332 Train Acc: 77.61% Test Loss: 1.358 Test Acc: 74.41% Valid Loss: 1.331 Valid Acc: 77.83%	EPOCH: 25 time elapsed: 29.637 Train Loss: 1.255 Train Acc: 86.22% Test Loss: 1.297 Test Acc: 81.63% Valid Loss: 1.255 Valid Acc: 86.27%	EPOCH: 45 time elapsed: 28.776 Train Loss: 1.233 Train Acc: 88.68% Test Loss: 1.278 Test Acc: 83.66% Valid Loss: 1.231 Valid Acc: 89.21%
dir_CIFAR10_train_ff0011 0/25/50	DEFAULT (False)	DEFAULT (False)	0.001	EPOCH: 10 time elapsed: 28.958 Train Loss: 1.260 Train Acc: 85.93% Test Loss: 1.318 Test Acc: 79.50% Valid Loss: 1.260 Valid Acc: 85.98%	EPOCH: 25 time elapsed: 28.704 Train Loss: 1.167 Train Acc: 95.38% Test Loss: 1.278 Test Acc: 83.85% Valid Loss: 1.167 Valid Acc: 95.45%	EPOCH: 49 time elapsed: 26.946 Train Loss: 1.134 Train Acc: 98.25% Test Loss: 1.267 Test Acc: 85.27% Valid Loss: 1.136 Valid Acc: 98.14%

Table 29: CIFAR10 (Learning Rate = 0.0001)

Directory	Exclusivity	Group Sparsity	Learning rate	Accuracies for Exp 0 (Best Run of 10)	Accuracies for Exp 0 (Best Run of 25)	Accuracies for Exp 0 (Best Run of 50)
dir_CIFAR10_train_ft0000110 /25/50	DEFAULT (False)	True	0.0001	EPOCH: 10 time elapsed: 30.147 Train Loss: 1.329 Train Acc: 78.63% Test Loss: 1.351 Test Acc: 75.78% Valid Loss: 1.328 Valid Acc: 78.75%	EPOCH: 25 time elapsed: 28.246 Train Loss: 1.230 Train Acc: 89.22% Test Loss: 1.288 Test Acc: 82.58% Valid Loss: 1.231 Valid Acc: 89.09%	EPOCH: 49 time elapsed: 31.059 Train Loss: 1.177 Train Acc: 94.69% Test Loss: 1.262 Test Acc: 85.52% Valid Loss: 1.179 Valid Acc: 94.52%
dir_CIFAR10_train_tt0000110 /25/50	True	True	0.0001	EPOCH: 10 time elapsed: 29.183 Train Loss: 1.339 Train Acc: 77.47% Test Loss: 1.360 Test Acc: 74.85% Valid Loss: 1.339 Valid Acc: 77.70%	EPOCH: 25 time elapsed: 29.313 Train Loss: 1.237 Train Acc: 88.56% Test Loss: 1.290 Test Acc: 82.57% Valid Loss: 1.239 Valid Acc: 88.21%	EPOCH: 47 time elapsed: 30.548 Train Loss: 1.203 Train Acc: 91.95% Test Loss: 1.275 Test Acc: 84.16% Valid Loss: 1.203 Valid Acc: 92.15%
dir_CIFAR10_train_tf0000110 /25/50	True	DEFAULT (False)	0.0001	EPOCH: 9 time elapsed: 28.744 Train Loss: 1.341 Train Acc: 77.70% Test Loss: 1.364 Test Acc: 75.04% Valid Loss: 1.342 Valid Acc: 77.87%	EPOCH: 24 time elapsed: 28.114 Train Loss: 1.215 Train Acc: 91.12% Test Loss: 1.281 Test Acc: 83.70% Valid Loss: 1.216 Valid Acc: 91.16%	EPOCH: 50 time elapsed: 29.052 Train Loss: 1.153 Train Acc: 96.85% Test Loss: 1.264 Test Acc: 85.31% Valid Loss: 1.156 Valid Acc: 96.41%
dir_CIFAR10_train_ff0000110 /25/50	DEFAULT (False)	DEFAULT (False)	0.0001	EPOCH: 9 time elapsed: 27.607 Train Loss: 1.349 Train Acc: 77.06% Test Loss: 1.367 Test Acc: 74.40% Valid Loss: 1.348 Valid Acc: 77.38%	EPOCH: 25 time elapsed: 27.143 Train Loss: 1.222 Train Acc: 90.27% Test Loss: 1.286 Test Acc: 83.02% Valid Loss: 1.226 Valid Acc: 89.84%	EPOCH: 49 time elapsed: 27.572 Train Loss: 1.154 Train Acc: 96.63% Test Loss: 1.259 Test Acc: 85.78% Valid Loss: 1.156 Valid Acc: 96.47%

Table 30: CIFAR10 (Learning Rate = 0.00001)

Directory	Exclusivity	Group Sparsity	Learning rate	Accuracies for Exp 0 (Best Run of 10)	Accuracies for Exp 0 (Best Run of 25)	Accuracies for Exp 0 (Best Run of 50)
dir_CIFAR10_train_ft000011 0/25/50	DEFAULT (False)	True	0.00001	EPOCH: 10 time elapsed: 28.835 Train Loss: 1.588 Train Acc: 52.62% Test Loss: 1.583 Test Acc: 53.75% Valid Loss: 1.586 Valid Acc: 53.03%	EPOCH: 25 time elapsed: 28.902 Train Loss: 1.484 Train Acc: 60.01% Test Loss: 1.484 Test Acc: 60.15% Valid Loss: 1.480 Valid Acc: 60.34%	EPOCH: 50 time elapsed: 29.429 Train Loss: 1.396 Train Acc: 71.88% Test Loss: 1.405 Test Acc: 70.78% Valid Loss: 1.392 Valid Acc: 72.17%
dir_CIFAR10_train_tt000011 0/25/50	True	True	0.00001	EPOCH: 10 time elapsed: 29.432 Train Loss: 1.588 Train Acc: 52.74% Test Loss: 1.582 Test Acc: 53.99% Valid Loss: 1.586 Valid Acc: 53.12%	EPOCH: 25 time elapsed: 29.631 Train Loss: 1.486 Train Acc: 59.87% Test Loss: 1.485 Test Acc: 59.78% Valid Loss: 1.481 Valid Acc: 60.10%	EPOCH: 50 time elapsed: 29.914 Train Loss: 1.395 Train Acc: 71.72% Test Loss: 1.405 Test Acc: 70.84% Valid Loss: 1.391 Valid Acc: 72.14%
dir_CIFAR10_train_tf000011 0/25/50	True	DEFAULT (False)	0.00001	EPOCH: 10 time elapsed: 29.394 Train Loss: 1.594 Train Acc: 52.16% Test Loss: 1.588 Test Acc: 53.52% Valid Loss: 1.592 Valid Acc: 52.57%	EPOCH: 24 time elapsed: 29.343 Train Loss: 1.498 Train Acc: 58.59% Test Loss: 1.496 Test Acc: 58.72% Valid Loss: 1.494 Valid Acc: 58.64%	EPOCH: 50 time elapsed: 29.124 Train Loss: 1.416 Train Acc: 69.00% Test Loss: 1.423 Test Acc: 68.56% Valid Loss: 1.412 Valid Acc: 69.34%
dir_CIFAR10_train_ff0000110 /25/50	DEFAULT (False)	DEFAULT (False)	0.00001	EPOCH: 10 time elapsed: 26.887 Train Loss: 1.596 Train Acc: 51.96% Test Loss: 1.590 Test Acc: 53.36% Valid Loss: 1.594 Valid Acc: 52.22%	EPOCH: 25 time elapsed: 27.551 Train Loss: 1.499 Train Acc: 58.08% Test Loss: 1.497 Test Acc: 58.45% Valid Loss: 1.495 Valid Acc: 58.27%	EPOCH: 50 time elapsed: 27.889 Train Loss: 1.425 Train Acc: 67.76% Test Loss: 1.431 Test Acc: 66.78% Valid Loss: 1.421 Valid Acc: 68.19%

The final step was to run the detection and accommodation algorithm on the new patterns. Table 31 and Table 32 provide the results of this phase. We used our model that was trained using the learning rate = 0.0001, exclusivity = false, and group sparsity = true. The best results for this phase were found when exclusivity = false, group sparsity = true, and the number of training epochs = 25.

Table 31: CIFAR10 Detection and Accommodation

Directory	Exclusivity	Group Sparsity	Accuracies for Exp 0 (Best Run of 50)	Accuracies for Exp 0 [0.0001, 13 epochs]	Accuracies for Exp 0 [0.0001, 25 epochs]
dir_CIFAR_train_ft000150 False, True	DEFAULT (False)	True	EPOCH: 49 time elapsed: 31.059 Train Loss: 1.177 Train Acc: 94.69% Test Loss: 1.262 Test Acc: 85.52% Valid Loss: 1.179 Valid Acc: 94.52%	Acc with 7 classes [9 5 2 4 7 1 0]: 77.33% Accuracy 8 - ship : [256] 46.8% Acc with 8 classes [9 5 2 4 7 1 0 8]: 39.42% Accuracy 6 - frog : [75] 85.0% Acc with 9 classes [9 5 2 4 7 1 0 8 6]: 11.35% Accuracy 3 - cat : [500] 0% Acc with all 10 classes [9 5 2 4 7 1 0 8 6 3]: 11.35%	Acc with 7 classes [9 5 2 4 7 1 0]: 77.84% Accuracy 8 - ship : [256] 46.8% Acc with 8 classes [9 5 2 4 7 1 0 8]: 39.39% Accuracy 6 - frog : [74] 84.4% Acc with 9 classes [9 5 2 4 7 1 0 8 6]: 35.48% Accuracy 3 - cat : [75] 85.0% Acc with all 10 classes [9 5 2 4 7 1 0 8 6 3]: 32.20%
dir_CIFAR10_train_tt000150 True, True	DEFAULT (False)	True	EPOCH: 47 time elapsed: 30.548 Train Loss: 1.203 Train Acc: 91.95% Test Loss: 1.275 Test Acc: 84.16% Valid Loss: 1.203 Valid Acc: 92.15%	Acc with 7 classes [9 5 2 4 7 1 0]: 71.41% Accuracy 8 - ship : [290] 42.0% Acc with 8 classes [9 5 2 4 7 1 0 8]: 41.02% Accuracy 6 - frog : [86] 82.8% Acc with 9 classes [9 5 2 4 7 1 0 8 6]: 11.35% Accuracy 3 - cat : [500] 0% Acc with all 10 classes [9 5 2 4 7 1 0 8 6 3]: 11.11%	Acc with 7 classes [9 5 2 4 7 1 0]: 71.41% Accuracy 8 - ship : [290] 42.0% Acc with 8 classes [9 5 2 4 7 1 0 8]: 41.06% Accuracy 6 - frog : [86] 82.4% Acc with 9 classes [9 5 2 4 7 1 0 8 6]: 35.64% Accuracy 3 - cat : [128] 74.4% Acc with all 10 classes [9 5 2 4 7 1 0 8 6 3]: 32.45%
dir_CIFAR10_train_tf000150 True, False	DEFAULT (False)	True	EPOCH: 50 time elapsed: 29.052 Train Loss: 1.153 Train Acc: 96.85% Test Loss: 1.264 Test Acc: 85.31% Valid Loss: 1.156 Valid Acc: 96.41%	Acc with 7 classes [9 5 2 4 7 1 0]: 83.61% Accuracy 8 - ship : [234] 53.2% Acc with 8 classes [9 5 2 4 7 1 0 8]: 41.02% Accuracy 6 - frog : [378] 24.4% Acc with 9 classes [9 5 2 4 7 1 0 8 6]: 23.49% Accuracy 3 - cat : [371] 25.8% Acc with all 10 classes [9 5 2 4 7 1 0 8 6 3]: 23.49%	Acc with 7 classes [9 5 2 4 7 1 0]: 83.61% Accuracy 8 - ship : [234] 53.2% Acc with 8 classes [9 5 2 4 7 1 0 8]: 12.60% Accuracy 6 - frog : [499] 0.2% Acc with 9 classes [9 5 2 4 7 1 0 8 6]: 12.60% Accuracy 3 - cat : [488] 2.4% Acc with all 10 classes [9 5 2 4 7 1 0 8 6 3]: 12.60%
dir_CIFAR10_train_ff000150 False, False	DEFAULT (False)	True	EPOCH: 49 time elapsed: 27.572 Train Loss: 1.154 Train Acc: 96.63% Test Loss: 1.259 Test Acc: 85.78% Valid Loss: 1.156 Valid Acc: 96.47%	Acc with 7 classes [9 5 2 4 7 1 0]: 85.29% Accuracy 8 - ship : [124] 75.20% Acc with 8 classes [9 5 2 4 7 1 0 8]: 64.29% Accuracy 6 - frog : [59] 88.20% Acc with 9 classes [9 5 2 4 7 1 0 8 6]: 20.21% Accuracy 3 - cat : [397] 20.6% Acc with all 10 classes [9 5 2 4 7 1 0 8 6 3]: 20.00%	Acc with 7 classes [9 5 2 4 7 1 0]: 85.29% Accuracy 8 - ship : [124] 75.20% Acc with 8 classes [9 5 2 4 7 1 0 8]: 48.04% Accuracy 6 - frog : [139] 72.20% Acc with 9 classes [9 5 2 4 7 1 0 8 6]: 20.02% Accuracy 3 - cat : [150] 70.0% Acc with all 10 classes [9 5 2 4 7 1 0 8 6 3]: 18.95%

Table 32: CIFAR10 Detection and Accommodation (Cont'd)

Directory	Exclusivity	Group Sparsity	Accuracies for Exp 0 (0.0001, 25 Epochs)
dir_CIFAR_train_ft000150 False, True	True	True	Acc with 7 classes [9 5 2 4 7 1 0]: 77.84% Accuracy 8 - ship : [256] 46.8% Acc with 8 classes [9 5 2 4 7 1 0 8]: 39.36% Accuracy 6 - frog : [80] 84.0% Acc with 9 classes [9 5 2 4 7 1 0 8 6]: 35.56% Accuracy 3 - cat : [75] 85.0% Acc with all 10 classes [9 5 2 4 7 1 0 8 6 3]: 32.25%
dir_MNIST_train_ft000150 False, True	True	Default (False)	Acc with 7 classes [9 5 2 4 7 1 0]: 77.84% Accuracy 8 - ship : [256] 46.8% Acc with 8 classes [9 5 2 4 7 1 0 8]: 39.38% Accuracy 6 - frog : [77] 84.6% Acc with 9 classes [9 5 2 4 7 1 0 8 6]: 35.70% Accuracy 3 - cat : [80] 84.0% Acc with all 10 classes [9 5 2 4 7 1 0 8 6 3]: 32.32%
dir_MNIST_train_ft000150 False, True	DEFAULT (False)	Default (False)	Acc with 7 classes [9 5 2 4 7 1 0]: 77.84% Accuracy 8 - ship : [256] 46.8% Acc with 8 classes [9 5 2 4 7 1 0 8]: 12.5% Accuracy 6 - frog : [500] 0% Acc with 9 classes [9 5 2 4 7 1 0 8 6]: 12.5% Accuracy 3 - cat : [500] 0% Acc with all 10 classes [9 5 2 4 7 1 0 8 6 3]: 12.5%

The best overall results achieved with CIFAR10 was 32.32%. At this point we question not only the extendibility of this process to other model architectures but also to other data sets (CIFAR10).

4.4.4 Summary.

We have been able to get the SHELS code to run and provide similar results as published for MNIST. The overall architecture is well thought out and there are several concepts that make sense. A weak link in their implementation is the determination of whether the new pattern is out of distribution or novel. Secondly, we question the extensibility of their approach. Our results did not seem to extend to CIFAR10 dataset. We could not get any of the generic models to run.

As we were using the code, we found it very difficult to understand (lack of comments) and in many cases it was hardcoded. We tried on several occasions to contact the developer with questions. We were successful once. On other occasions our emails were not answered. Based on our experience we believe the code will have to be re-written if we wish to go further and, in several cases, redesigned. It is for these reasons we are looking at other alternatives.

4.5 OpenCon: Open-world Contrastive Learning

Up until now we have presented several models that are based on many common ideas. They all were trained on a closed set of labeled data and used a gradient descent algorithm to learn/adjust the weights. They also used a biologically inspired architecture of interconnected neurons. In our last section we describe a new approach to machine learning, Open-world Contrastive Learning or OpenCon. OpenCon is described in detail in the paper, “OpenCon: Open-world Contrastive Learning” written by Yiyou Sun and Yizuan Li, University of Wisconsin-Madison [26]. The code is available at: <https://github.com/deeplearning-wisc/opencon>. The code ran as is. There were no changes required to get it running. We provide a short summary of OpenCon below followed by details of what we have done under this effort.

OpenCon is based on contrastive learning. It is designed to produce a compact representation space for both known and novel classes along with facilitating novelty discovery. The model is trained on a labeled and an unlabeled dataset (with samples from both known and novel classes). Traditional or existing training methods are not designed to simultaneously deal with both types of data. The authors provided a few examples: Supervised Contrastive Learning only assumes a labeled dataset without considering the unlabeled one; Weakly Supervised Contrastive Learning assumes the same classes in both the labeled and unlabeled dataset; and Self-supervised learning relies completely on the unlabeled dataset.

The goal of OpenCon is to learn representations for both datasets simultaneously. They begin by initializing several prototypes (generally equal to the number of potential classes) and evolve them based on a learning scheme. They also use the prototype vectors to perform out-of-distribution (OOD) detection (i.e., known vs unknown or novel data). For each sample they measure the cosine similarity between its representation and each prototype vector. If the sample embedding is far away from all the known class prototypes, then it is considered to be novel. Figure 55 provides a summary of their process as presented in their paper.

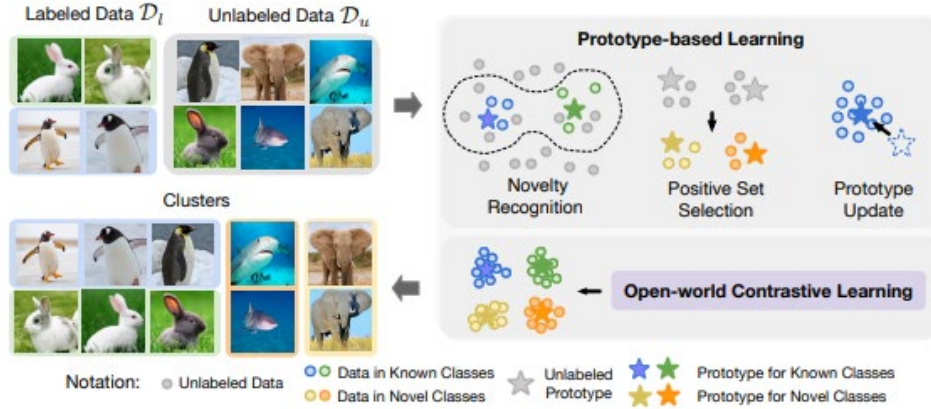


Figure 55: Overview of Learning Framework (author's paper, page 2 in [26])

There are two concepts that we need to discuss before continuing. The first is what they call Open-world Contrastive Loss. This loss function is used as core to their learning framework. It is defined as a function of the contrastive loss for the novel data plus the supervised loss on the labeled data plus the self-contrastive loss. The equation below (taken from their paper) defines this overall loss function.

$$\mathcal{L}_{\text{OpenCon}} = \lambda_n \mathcal{L}_n + \lambda_l \mathcal{L}_l + \lambda_u \mathcal{L}_u,$$

The second important concept is the updating of the prototype vectors. The prototypes are updated in a moving average style. μ_c (also taken from the paper) is the prototype of class c . The amount of change each time is determined by gamma or the momentum term. We will be getting back to this idea in a later section.

$$\mu_c := \text{Normalize}(\gamma \mu_c + (1 - \gamma) \mathbf{z}), \text{ for } c = \begin{cases} y \text{ (ground truth label)}, & \text{if } \mathbf{z} \in \mathcal{D}_l \\ \text{argmax}_{j \in \mathcal{Y}_n} \mu_j^\top \cdot \mathbf{z}, & \text{if } \mathbf{z} \in \mathcal{D}_n \end{cases}$$

OpenCon's contribution is the code that sits on top of the backbone model. A backbone model is an existing model developed by another researcher and pretrained using a given dataset. For example, OpenCon currently is configured to use the SimCLR trained model (developed by Chen, Kornblith, Norouzi and Hinton and described in their paper entitled "A Simple Framework for Contrastive Learning of Visual Representations"). The simclr_cifar_100 model is trained with ResNet18 on the CIFAR-100 dataset while simclr_imagenet_100 uses ResNet50 and the ImageNet dataset. A projection layer sits on top of a given backbone model and is trained using the OpenCon Learning Framework. Figure 56 displays the overall architecture of OpenCon.

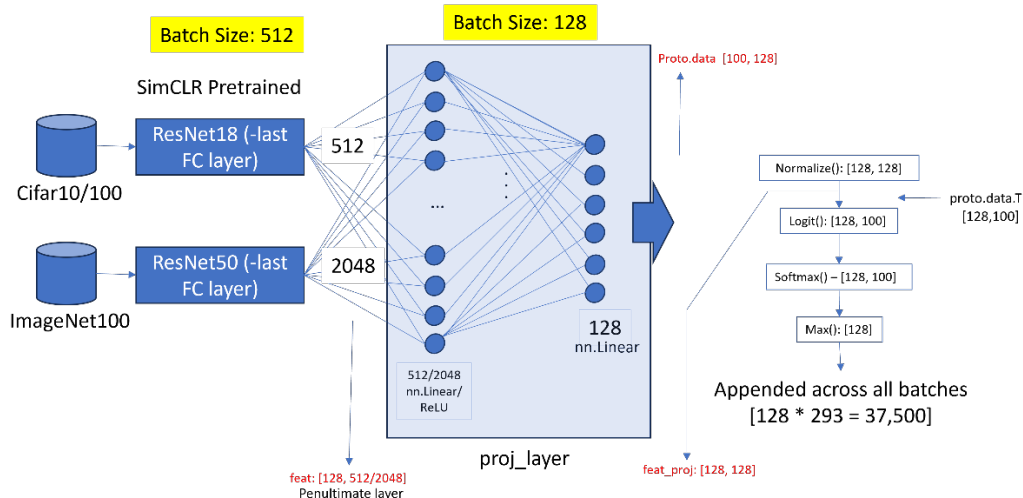


Figure 56: Baseline OpenCon Architecture

By default, classes are divided into 50% seen and 50% novel. 50% of the known classes are considered as the labeled dataset and the rest as the unlabeled. Table 33 provides a summary of the hyper-parameters used for both CIFAR-100 and ImageNet-100.

Table 33: Hyper- Parameters

Parameter	Value
Training Epochs	200 (CIFAR-100)/ 120 (ImageNet-100)
Batch Size	512
Training Algorithm	Stochastic Gradient Descent (SGD)
Momentum	Learning Rate Momentum 0.9
Weight Decay	10-4
Learning Rate	Starts at 0.02, decays by a factor of 10 at 50% and the 75% of training stage
Momentum – Prototype	0.9
Percentile (p) for OOD	70%
KL-divergence regularizer	0.05

Table 34 provides a summary of the results documented in [26]. Our goal initially was to get the code to run within Windows 11 and under the PyCharm IDE.

Table 34: Baselined Test Results (taken from paper)

Method	CIFAR-100			ImagNet-100		
	All	Novel	Seen	All	Novel	Seen
[†] FixMatch (Kurakin et al., 2020)	20.3	23.5	39.6	34.9	36.7	65.8
[†] DS ³ L (Guo et al., 2020)	24.0	23.7	55.1	30.8	32.5	71.2
[†] CGDL (Sun et al., 2020)	23.6	22.5	49.3	31.9	33.8	67.3
*DTC (Han et al., 2019)	18.3	22.9	31.3	21.3	20.8	25.6
*RankStats (Zhao & Han, 2021)	23.1	28.4	36.4	40.3	28.7	47.3
*SimCLR (Chen et al., 2020a)	22.3	21.2	28.6	36.9	35.7	39.5
ORCA (Cao et al., 2022)	47.2 $\pm$ 0.7	41.0 $\pm$ 1.0	66.7 $\pm$ 0.2	76.4 $\pm$ 1.3	68.9 $\pm$ 0.8	89.1 $\pm$ 0.1
GCD (Vaze et al., 2022)	46.8 $\pm$ 0.5	43.4 $\pm$ 0.7	69.7$\pm$0.4	75.5 $\pm$ 1.4	72.8 $\pm$ 1.2	90.9 $\pm$0.2
OpenCon (Ours)	52.7$\pm$0.6	47.8$\pm$0.6	69.1 $\pm$ 0.3	83.8$\pm$0.3	80.8$\pm$0.3	90.6 $\pm$ 0.1

To get the OpenCon code to run was very easy. All we had to do was to configure the environment. We decided to use Python 3.7, Pytorch 1.13.1 with Cuda 11.6. We ran both examples after we downloaded the simClr pretrained weights for CIFAR-100 and IMAGENET-100 and the ImagNet dataset. The models successfully ran and produced very similar results to those presented in the paper (Table 35). The only problem we had was the amount of GPU memory required to run the ImagNet100 model. We could not run it using our GPU co-processor with 4 GBs residing on our laptop. We had to use an external GeForce RTX board with 24GBs. It still took quite a while to run.

Table 35: Test Results

Dataset	Overall Accuracy	Unseen Accuracy	Seen Accuracy
CIFAR-100	0.5170	0.4644	0.6919
ImagNet-100	0.8324	0.7959	0.9066

The problem we saw was that even though we were able to run the ImagNet-100 model via the external board it was extremely slow. After discussions with several people, we found that the problem of using an external chassis (which housed the board) was connected with a USB-C, Thunderbolt interface. This interface limited use to 10GBs transfers. The decision was to configure/build a low-cost pc where everything was in the same box. We did this. For the CIFAR-100 model it took 06:18:29.56 to run on the laptop. To run on the all-in-one desktop unit, it took 01:40:26.94 – which yielded a performance improvement of 3.8 times.

4.5.1 Modifications made to OpenCon codebase.

Once we got the model up and running several questions were raised. In this section we summarize these and what we did to answer them. These questions include:

- Can we add a layer in the projection head? Does it affect overall performance?
- How easy is it to change the backbone model (from ResNet18 or 50 to a transformer model, DINO_V2)?

- Does the distribution we use to initialize the prototypes make a difference?
- Does the number of output features make a difference?
- Can we save the ‘best’ model for later use and what does ‘best’ mean?
- Can we fix the initial values of the prototypes?
- How does the momentum term affect the system to learn?
- Can we save/reload a trained model and use a separate test data set? What are the results?

4.5.1.1 Effect of Adding a Layer.

The first question we looked at was the effect of adding a layer to the projection head. Figure 57 shows the revised architecture. The layer size we added was 2048. We used the CIFAR-100 model/dataset.

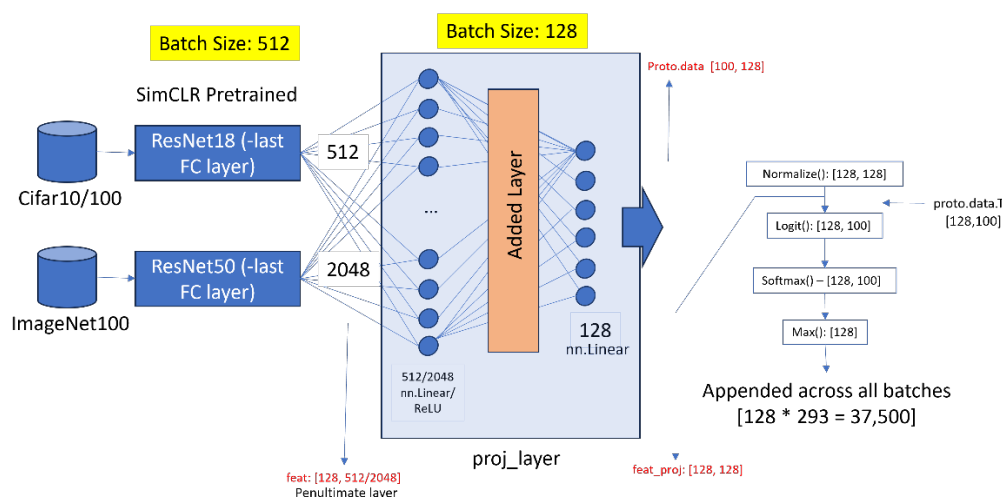


Figure 57: OpenCon Architecture after Adding Layer

Table 36 provides the results. The addition of a layer provided a decrease in accuracy for both seen and unseen patterns.

Table 36: Test Results (After Adding Layer)

Dataset	Overall Accuracy	Unseen Accuracy	Seen Accuracy
Baseline (Paper)	0.527	0.478	0.691
Added middle layer	0.4758	0.4271	0.7058

4.5.1.2 Changing the Back-end Model.

In this next section we looked at the effect of changing the back-end model to DINO_V2 for CIFAR-100 and ImageNet-100. Figure 58 displays our augmented model.

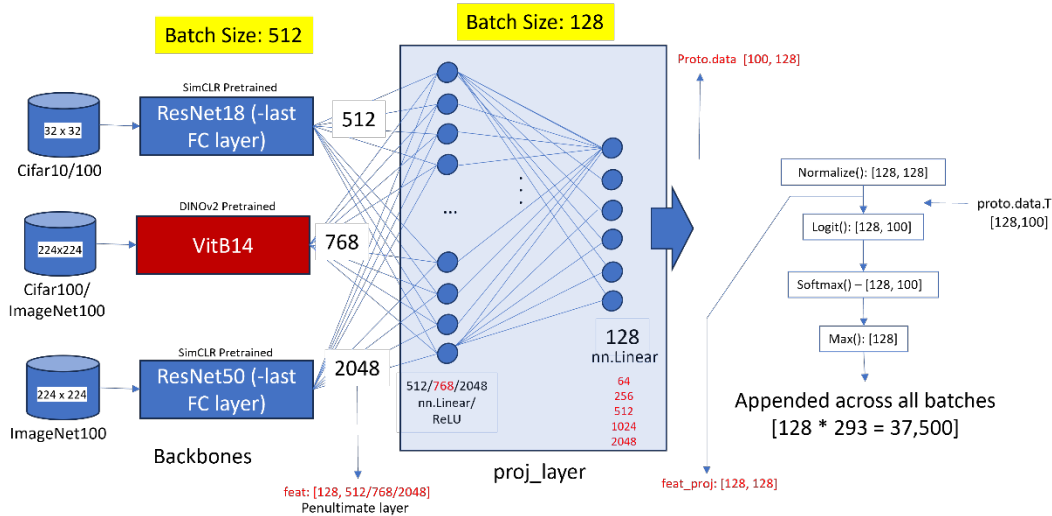


Figure 58: OpenCon Architecture including additional backbone network (DINO_V2)

We began by looking at the DINO_V2 model. DINO_V2 is a transformer model. Transformer models were originally used for text understanding and then applied to images. Figure 59 (provided in a paper by Cameron R. Wolfe, entitled “Using Transformers for Computer Vision”) provides a basic depiction of a vision transformer architecture.

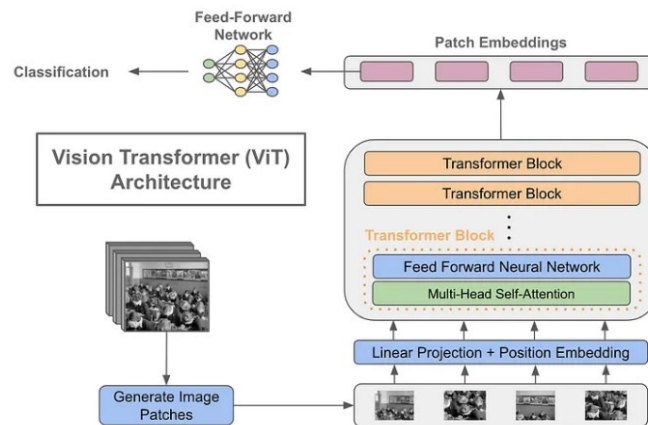


Figure 59: Basic Vision Transformer Architecture

Images are divided into several patches and then sent onto the learning or self-attention algorithms. We have taken a pre-trained version of DINO Version 2 and integrated it as part of our overall model. There were a few changes made to the code for this to happen:

- Both models (CIFAR-100 and ImageNet-100) needed to pre-load the trained DINO_V2 model. We used the ViTB14 version trained on ImageNet. The ViTB14 version uses the

‘Big’ patch size of 14x14. This means that the image must be evenly divisible by 14 in both directions (hence the image size was chosen to be 224).

- CIFAR-100 was using 32x32 size images. We needed to scale the images up to use model.
- ImageNet-100 was much easier to add than CIFAR-100. It did not require any changes in the images (already 224 x 224)

Table 37 provides the results. The original OpenCon used DINO with ImageNet-100 and got pretty good results. We used DINO_V2 and achieved even better overall results for both CIFAR-100 and ImageNet-100. The only problem is the amount of time it took to train the models. For example, the CIFAR-100 model with ResNet18 took only 1 hour 26 minutes to train. With DINO_V2 it took a little of a day.

Table 37: Test Results - Using DINO_V2

Dataset	Overall Accuracy	Unseen Accuracy	Seen Accuracy	Elapsed Time
Baseline (Paper – CIFAR-100)	0.527	0.478	0.691	1:26:00
Using DINO_V2/CIFAR-100	0.7397 (↑0.2127)	0.6934 (↑0.2154)	0.9050 (↑0.2140)	1 day, 3:39:46
Baseline (Paper – ImageNet-100)	0.838	0.808	0.906	-
Baseline with DINO	0.840	0.812	0.938	-
Using DINO_V2/ImageNet-100	0.8953 (↑0.0513)	0.8706 (↑0.0586)	0.9545 (↑0.0165)	1 day, 19:49:40

4.5.1.3 Initializing the Prototypes.

We were next asked if the initialization of the prototypes made a difference. The baseline code used a normal distribution. We modified this distribution to look at a uniform distribution. We also added several probes for saving the initial prototypes in a .mat file (MATLAB) for further analysis. We reran our CIFAR-100 model and compared the results (Table 38):

Table 38: Test Results - Investigating the Initialization Distribution of the Prototypes

Dataset	Overall Accuracy	Unseen Accuracy	Seen Accuracy
Baseline (Paper)	0.527	0.478	0.691
Normal Distribution	0.5170	0.4644	0.6919
Uniform Distribution	0.5296	0.4764	0.6882

The next question that we investigated was “What was the effect of the number of output features (CIFAR-100) on overall results?”. To answer this, we ran the model several times altering the number of output features for 64 to 2048 outputs. Table 39 provides a summary of the results.

Table 39: Test Results - Effect of Output Feature Size

Number of Features	Overall Accuracy	Seen Accuracy	Unseen Accuracy
64	0.5194	0.6893	0.4754
128	0.5241	0.6919	0.4740
256	0.5123	0.6928	0.4560
512	0.4818	0.7061	0.4284
1024	0.4364	0.7153	0.4010
2048	0.3703	0.7350	0.3284

Looks like seen accuracy does increase as the number of output features also increase but overall and unseen accuracies decrease. Seems that the authors choose wisely - the optimal output size of 128.

4.5.1.4 Saving the ‘Best’ Configuration.

OpenCon currently writes out the weights at predetermined/preset epochs (for CIFAR-100: 99 and 199). Other models write out weights based on some ‘best’ measure, e.g., testing or validation accuracy. What is the measure that OpenCon can use for ‘best’: Overall Accuracy, Seen Accuracy or Unseen Accuracy or something else (Mean Uncertainty)? We ran our baseline CIFAR-100 model. Figure 60 provides a chart summarizing the run. Overall and Unseen accuracies peak at the same time (194). Seen accuracy is .005 from its peak (27 epochs). However, at 27 epochs overall accuracy is down 0.0525 and unseen accuracy is down 0.0414 from its peak at epoch 194. We decided to use Overall Accuracy.

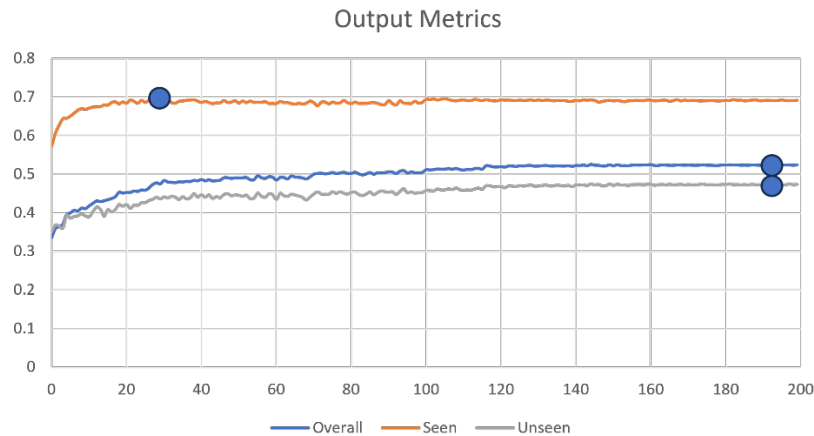


Figure 60: Investigating Collection Metrics

4.5.1.5 Effect of the Momentum Term.

Our next question was “What was the effect of changing the size of the momentum term?” We were curious as to how sensitive this parameter was and what happens if we fixed the prototype at the initial values, not allowing them to change. In the following sections we investigate this question for momentum terms: 0.0, 0.9, 0.99, 0.99 and 1.0. In the following subsections we

provide for each momentum value the best overall results, how long it took to complete the run, the minimum, maximum and average separation between the initial prototype value to its final, i.e., the spread. We also provide two charts: (1) initial vs final placement/values of the prototypes and (2) the clusters of the learned patterns.

- Momentum = 0.9 (Baseline), Figure 61.
 - Best Overall values found at Epoch: 194 Overall Accuracy: 0.5257, Seen Accuracy: 0.6919, Unseen Accuracy: 0.4740.
 - Training Completed, Elapsed Time: 1:26:11.7.
 - Deltas (changes between initial and final) - Min: 1.0667 Max: 1.3736 Mean: 1.221.

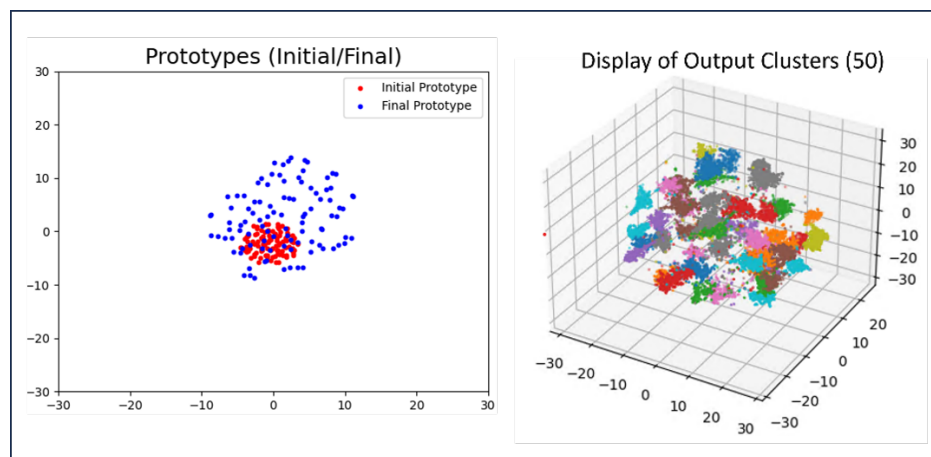


Figure 61: Momentum = 0.9 (Baseline)

- Momentum = 0.99, Figure 62.
 - Best Overall values found at Epoch: 197 Overall Accuracy: 0.5179, Seen Accuracy: 0.6912, Unseen Accuracy: 0.4670.
 - Training Completed, Elapsed Time: 1:25:52.16.
 - Deltas (changes between initial and final) - Min: 0.1310 Max: 1.3896 Mean: 1.18835.

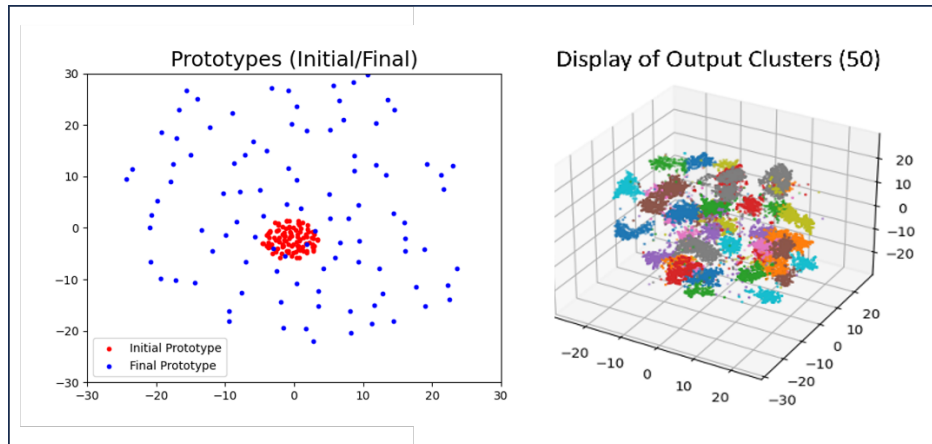


Figure 62: Momentum = 0.99

- Momentum = 0.999, Figure 63.
 - Best Overall values found at Epoch: 170 Overall Accuracy: 0.5157, Seen Accuracy: 0.6883, Unseen Accuracy: 0.4589.
 - Training Completed, Elapsed Time: 1:26:34.89.
 - Deltas (changes between initial and final) - Min: 0.2673 Max: 1.3761 Mean: 1.1781.

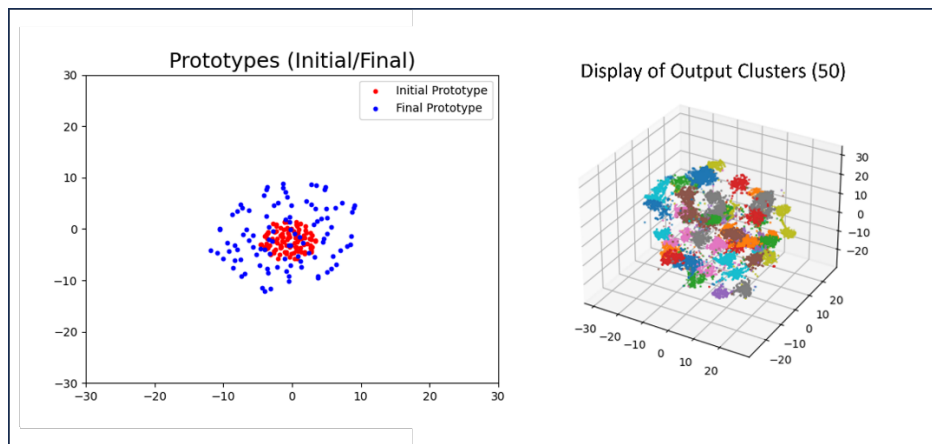


Figure 63: Momentum = 0.999

- Momentum = 1.0, Figure 64.
 - Best Overall values found at Epoch: 196 Overall Accuracy: 0.2671, Seen Accuracy: 0.0000, Unseen Accuracy: 0.3349.
 - Training Completed, Elapsed Time: 1:32:57.946238.

- Deltas (changes between initial and final) - Min: 0.0 Max: 4.742e-05 Mean: 9.2331e-06.

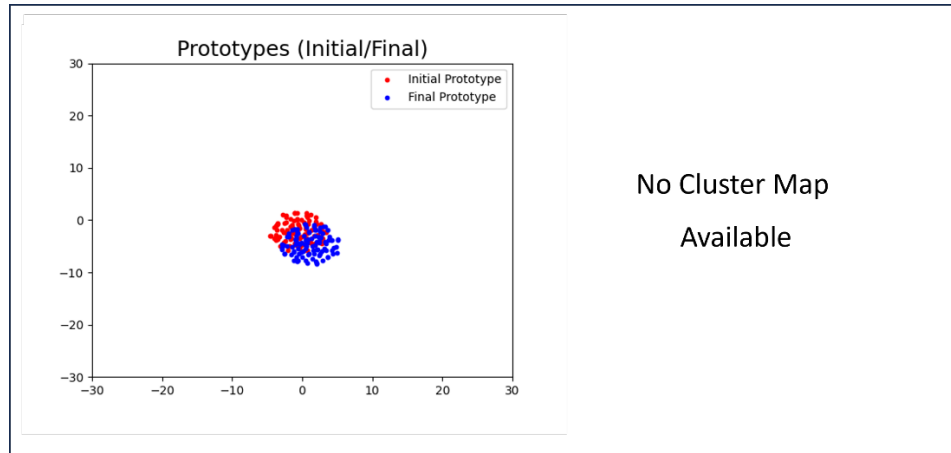


Figure 64: Momentum = 1.0

- Momentum = 0. Figure 65.
 - Best Overall values found at Epoch: 126 Overall Acc: 0.5066, Seen Acc: 0.6893, Unseen Acc: 0.4394.
 - Training Completed, Elapsed Time: 1:25:52.633841.
 - Deltas (changes between initial and final) - Min: 1.0597 Max: 1.4133 Mean: 1.2515.

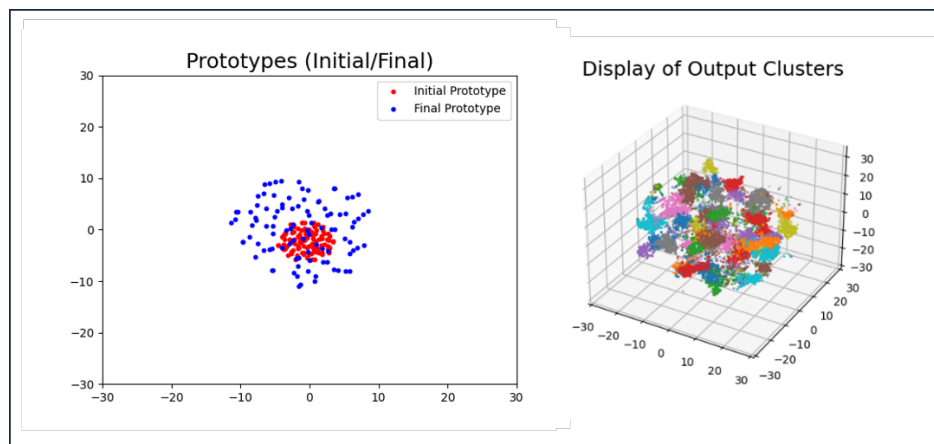


Figure 65: Momentum = 0.0

It seems that after looking at each of the runs, the momentum term matters in the overall learning process. The best results were achieved using the baseline momentum of 0.9.

4.5.1.6 Using a Separate Test Dataset.

OpenCon does not use a separate test set. We noticed while reviewing the code base OpenCon uses only the training set and divides it into two (50% labeled and 50% unlabeled). It uses the same data set for testing accuracy (Overall, seen and unseen). Refer to Figure 66. This is not normal. Generally, one maintains a separate test set that the model has not been trained on to compute overall accuracy. CIFAR-100 does have a test set that they do not use. We decided to see how the model did on this separate data set – never seen.

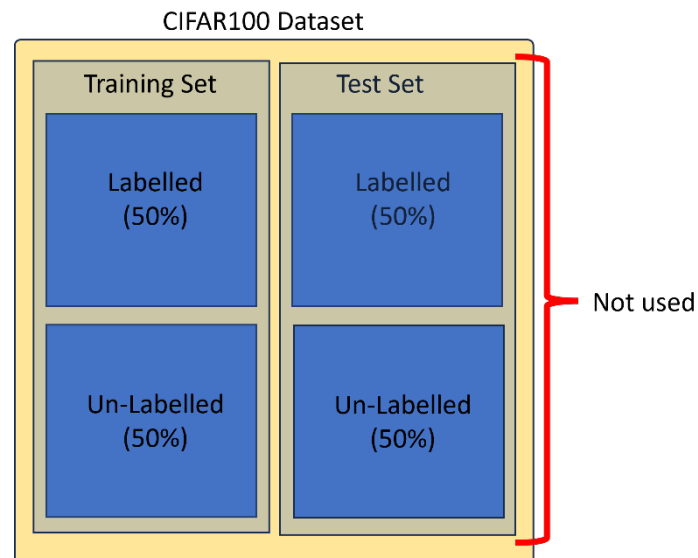


Figure 66: OpenCon CIFAR-100 Dataset

We loaded the data set using the same code that randomly split the data set in two. The overall results were poor.

Test overall acc 0.0499, seen acc 0.0082, unseen acc 0.0852

We wondered what was happening. We tried running the original dataset through our code. We could not reproduce our original results.

Test overall acc 0.0579, seen acc 0.0102, unseen acc 0.0844

The more we investigated the problem we found out that OpenCon does not store the fully trained model. It only has the original back-bone model and some state information but not the complete model. To use the model after being trained one needs to be able to save and load the resultant model. This is not currently possible and is recommended to be the next step.

4.5.2 Summary.

OpenCon is a new and exciting approach to model learning. It has the advantage of being able to handle both labeled and un-labeled datasets which make it more practical to the real world. However, as we can see, there are many questions and much work that needs to be done to understand it and use it. We also need to address lifelong learning in these models.

5.0 CONCLUSIONS

In this report we presented several machine learning models/techniques. We started with a unique model based on how people smell and tried to apply it to understanding images/objects. We found several limitations and based on what we saw changed direction to what others had, the CNN. We looked at a number of enhancements proposed (e.g., architectures, learning, optimizers) over the years that reduced overall error and at the same time increased their overall recognition. But CNNs themselves have evolved over time (from Lenet-5 to AlexNet to ResNet and beyond). We investigated several alternatives including random networks, kernel networks and those that address life-long learning (SHELS) and open-world scenarios (labeled and unlabeled data, including novel unseen patterns), like OpenCon. New models/architectures/ techniques continue to be proposed/developed. At this point the only thing we can say for certain is that the models we are using now will not be the models of tomorrow. Techniques/architectures continue to evolve, new ideas continue to be proposed/discovered and their power and abilities continue to grow.

6.0 REFERENCES

1. Nabil Imam, Thomas A. Cleland, Rapid online learning and robust recall in a neuromorphic olfactory circuit, Nat. Mach. Intell 2, 181–191. 10.1038/s42256-020-0159-4, <https://arxiv.org/ftp/arxiv/papers/1906/1906.07067.pdf>, 2020.
2. Yosh Bohra, “The Challenge of Vanishing/Exploding Gradients in Deep Neural Networks”, Updated On June 23rd, 2021, <https://www.analyticsvidhya.com/blog/2021/06/the-challenge-of-vanishing-exploding-gradients-in-deep-neural-networks/>.
3. MNIST (Modified National Institute of Standards and Technology) *dataset*, <https://www.kaggle.com/datasets/hojjatk/mnist-dataset>, 2018.
4. CIFAR-10/100 (Canadian Institute For Advanced Research) dataset, 2009, <https://www.cs.toronto.edu/~kriz/cifar.html>.
5. ImageNet dataset, ImageNet Large Scale Visual Recognition Challenge (ILSVRC) 2012-2017, Updated March 11, 2021, <https://www.image-net.org/challenges/LSVRC/index.php>.
6. Hamed F. Langroudi, Cory Merkel, Humza Syed, Dhireesha Kudithipudi, Exploiting Randomness in Deep Learning Algorithms, 19 July 2019, International Joint Conference on Neural Networks. Budapest, Hungary.
7. Alex Krizhevsky, Ilya Sutskever, Geoffrey E. Hinton, “ImageNet Classification with Deep Convolutional Neural Networks”, Advances in Neural Information Processing Systems 25, 2012.
8. Karen Simonyan, Andrew Zisserman, “Very Deep Convolutional Networks for Large-Scale Image Recognition”, [arXiv:1409.1556](https://arxiv.org/abs/1409.1556), 2014.
9. Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun, “Deep Residual Learning for Image Recognition”, [arXiv:1512.03385](https://arxiv.org/abs/1512.03385), 10 Dec 2015.

10. Rana Singh, “T-SNE visualization of high dimension MNIST dataset”, <https://ranasinghiitkgp.medium.com/t-sne-visualization-of-high-dimension-mnist-dataset-48fb23d1bafd>, 5 September 2019.
11. Geethan Karunaratne, Manuel Schmuck, Manuel Le Gallo, Giovanni Cherubini, Luca Benini, Abu Sebastian, Abbas Rahimi, Robust High-dimensional Memory-augmented Neural Networks, [arXiv:2010.019](https://arxiv.org/abs/2010.019), 19 March 2021.
12. Antonello Rosato; Massimo Panella; Evgeny Osipov; Denis Kleyko, “Few-shot Federated Learning in Randomized Neural Networks via Hyperdimensional computing”, 2022 International Joint Conference on Neural Networks (IJCNN), 18-23 July 2022.
13. Thimal Kempitiya; Daswin De Silva; Sachin Kahawala; Dilantha Haputhanthri; Daminda Alahakoon; Evgeny Osipov , “Parameterization of Vector Symbolic Approach for Sequence Encoding Based Visual Place Recognition”, 2022 International Joint Conference on Neural Networks (IJCNN), 18-23 July 2022.
14. Robert Guirado, Abbas Rahimi, Geethan Karunaratne, Eduard Alarcón, Abu Sebastian, Sergi Abadal , “Wireless on-Chip Communications for Scalable In-memory Hyperdimensional Computing”, [arXiv:2205.10889](https://arxiv.org/abs/2205.10889), 22 May 2022.
15. P Łuczak, K Ślot, J Kucharski, “Combining Deep Convolutional Feature Extraction with Hyperdimensional Computing for Visual Object Recognition”, 2022 International Joint Conference on Neural Networks (IJCNN), 2022.
16. Hadi Alasti, “Efficient tracking of spatially correlated signals in wireless sensor fields: A weighted stochastic gradient approach”, IET Digital Library, Volume 11, Issue 2, p.78 – 90, April 2021.
17. Hadi Alasti, “Communication-Efficient Tracking of Unknown, Spatially Correlated Signals in Ad-Hoc Wireless Sensor Networks: Two Machine Learning Approaches”, *Sensors* 2021, 21(15), 5175, 30 July 2021.
18. Dilantha Haputhanthri; Evgeny Osipov; Sachin Kahawala; Daswin De Silva; Thimal Kempitiya; Daminda Alahakoon, “Evaluating Complex Sparse Representation of Hypervectors for Unsupervised Machine Learning”, 2022 International Joint Conference on Neural Networks (IJCNN), 18-23 July 2022.
19. Shyu Duan, “Unifying Kernel Methods and Neural Networks and Modularizing Deep Learning”, PhD Dissertation, University of Florida, 2020.
20. Yarín Gal, Zoubin Ghahramani, “Dropout as a Bayesian Approximation: Representing Model Uncertainty in Deep Learning”, Proceedings of the 33 rd International Conference on Machine Learning, New York, NY, USA, 2016.
21. Daily Milanés-Hermosilla, Rafael Trujillo Codorníu, René López-Baracaldo, Roberto Sagaró-Zamora, Denis Delisle-Rodriguez, John Jairo Villarejo-Mayor, and José Ricardo Núñez-Álvarez, “Monte Carlo Dropout for Uncertainty Estimation and Motor Imagery Classification”, 21(21): 7241, November 2021.
22. Nok Chan, “Uncertainty estimation for Neural Network — Dropout as Bayesian Approximation”, Towards Data Science, Jan 28, 2019.

23. Ismail Alarab, Simant Prakoonwit, Mohamed Ikbali Nacer, “Illustrative Discussion of MC-Dropout in General Dataset: Uncertainty Estimation in Bitcoin”, Neural Processing Letters, 53(2):1001-1011, 2021.
24. Mehedi Hasan, Abbas Khosravi, Ibrahim Hossain, Ashikur Rahman, Saeid Nahavandi “Controlled Dropout for Uncertainty Estimation”, [arXiv:2205.03109](https://arxiv.org/abs/2205.03109), 6 May 2022.
25. Meghna Gummadi, David Kent, Jorge A. Mendez, Eric Eaton, “SHELS: Exclusive Feature Sets for Novelty Detection and Continual Learning Without Class Boundaries “, [arXiv:2206.13720](https://arxiv.org/abs/2206.13720), 28 Jun 2022.
26. Yiyu Sun, Yixuan Li, “OpenCon: Open-world Contrastive Learning”, 8 Jan 2023.

7.0 LIST OF ACRONYMS

AD	Apical Dendrite
AFRL	Air Force Research Laboratory
AlexNet	Alex (Krizhevsky’s) Network
AUC	Area Under the Curve
BP	Backpropagation
CIFAR 10/100	Canadian Institute for Advanced Research, 10/100 classes
CMC	Competitive Multi-scale Convolution
CNN	Convolutional Neural Network
CoRN	Competitive Residual Network
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
DMLP	Deep Multi-Layer Perceptron
DQN	Deep Q-Network
E2E	End-To-End
ELU	Exponential Linear Unit
EO	Electro-Optical
EPL	External Plexiform Layer
GC	Granule Cell
GFS	Government Furnished Software
GPU	Graphical Processing Unit
GUI	Graphical User Interface
HD	Hamming Distance
HPC	High-Performance Computing
IDE	Integrated Development Environment
IND	In-Distribution
KD Tree	K-Dimensional Tree
LeNet-5	LeCun’s Network
LReLU	Leaky Rectified Linear Unit

LSTM	Long Short-Term Memories
MAE	Mean Absolute Error
MAPE	Mean Absolute Percentage Error
MC	Mitral Cell
MNIST	Modified National Institute of Standards and Technology
MSE	Mean Square Error
MSTAR	Moving and Stationary Target Acquisition and Recognition
NaN	Not a Number
OOD	Out-Of-Distribution
OpenCon	Open World Contrastive Learning
PCA	Principal Component Analysis
PI	Principal Investigator
PPO	Proximal Policy
PReLU	Parametric Rectified Linear Unit
ReLU	Rectified Linear Unit
ResNet	Residual Network
RGB	Red Green Blue
RL	Re-enforcement Learning
RMS	Root Mean Square
SAR	Synthetic Aperture Radar
SELU	Scaled Exponential Linear Unit
SGD	Stochastic Gradient Descent
SHELS	Sparse High-Level-Exclusive, Low-level-Shared
simCLR	Simple Framework for Contrastive Learning of Visual Representations
SOM	Self-Organizing Map
STDP	Spike Timing-Dependent Plasticity
T-SNE	t-Distributed Stochastic Neighbor Embedding
VGG-19	Visual Geometry Group – 19
WRN	Wide Residual Network
WSG	Weighted Stochastic Gradient
XE	Cross Entropy