HARD ASH: SPARSITY AND THE RIGHT OPTIMIZER MAKE A CONTINUAL LEARNER

Santtu Keskinen
Unaffiliated
santtu.keskinen@gmail.com

ABSTRACT

In class incremental learning, neural networks typically suffer from catastrophic forgetting. We show that an MLP featuring a sparse activation function and an adaptive learning rate optimizer can compete with established regularization techniques in the Split-MNIST task. We highlight the effectiveness of the Adaptive SwisH (ASH) activation function in this context and introduce a novel variant, Hard Adaptive SwisH (Hard ASH) to further enhance the learning retention.

1 INTRODUCTION

Continual learning presents a unique challenge for artificial neural networks, particularly in the class incremental setting (Hsu et al., 2019), where a single network must remember old classes that have left the training set. In this paper I explore an overlooked approach that doesn't require any techniques developed specifically for continual learning. For regularization I used only carefully tuned optimizers with adaptive learning rate such as Adagrad (Duchi et al., 2011). The approach does not exploit the task structure in any way. This is in contrast to most regularizing continual learning methods that require either explicit task boundaries such as EWC (Kirkpatrick et al., 2017) and MAS (Aljundi et al., 2018) or implied task boundaries like Online EWC (Schwarz et al., 2018). Perhaps closest to my method are the Elephant MLP (Lan & Mahmood, 2023) and SDMLP (Bricken et al., 2023), but the results here outperform both in Split-MNIST with an arguably simpler method.

Sparse representations have been shown to be effective at reducing forgetting in neural networks (Srivastava et al., 2013; Shen et al., 2021; Ahmad & Scheinkman, 2019; Lan & Mahmood, 2023). I continue this pattern and show that combining sparsity with an adaptive learning rate optimizer is enough to make a conceptually simple but surprisingly effective continual learner.

To make my MLP hidden layer representations sparse, I used an activation function that makes the majority of the activations zero. Top-K (also known as k-WTA) is the conceptually simplest sparse activation function and it's usage in neural networks goes back to at least Makhzani & Frey (2014). I show that Top-K works well in my setup, but I get better accuracy with my novel Hard Adaptive Swish (Hard ASH) activation.

2 ASH AND HARD ASH

The Adaptive SwisH (ASH) activation function (Lee et al., 2022), introduced a new way of controlling the amount of sparsity of the activations, that is cheaper to compute than the Top-K function. This is the first study to use ASH for continual learning. The formulation for ASH I use is:

$$ASH(x_i) = x_i \cdot S(\alpha \cdot (x_i - \mu_X - z_k \cdot \sigma_X)), X = [x_1, x_2, \ldots, x_n]$$

S is the sigmoid function, μ_X and σ_X are the mean and standard deviation of the vector X, α is an hyperparameter that controls the slope of the sigmoid and z_k is a hyperparameter that controls the amount of sparsity. A higher z_k value corresponds to more sparsity in the activations.

2.1 HARD ASH

Lan & Mahmood (2023) theorized that activation functions should be better suited for continual learning if their gradients are fairly sparse, i.e. the activation function should be flat in most places. To reduce the gradient flow I replaced the sigmoid function with a hard sigmoid (Courbariaux et al., 2016) and clip the first x_i term to values between 0 and 2. See appendix A.2 for the exact Hard ASH formula.

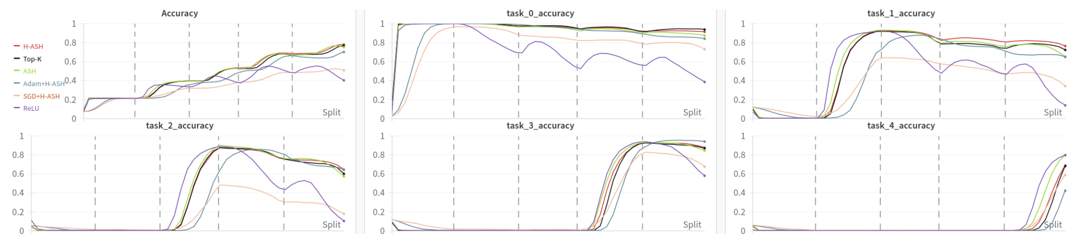

Figure 1: Overall and per-task validation accuracies of a single run of each method. Vertical lines represent the points in the training where the task changes. Optimizer is Adagrad when not specified. Best methods slowly lose accuracy on old tasks, but struggle to learn the last task. ReLU forgets the old tasks even with good optimizer like Adagrad. Meanwhile Hard ASH keeps some old-task performance even with plain SGD. Variations between runs are small enough to be barely visible.

3 Experiment

I ran an experiment on 5 task Split-MNIST dataset in the class incremental setting, the hardest setting of the Split-MNIST, where a single network has to learn all the tasks without task id input (Hsu et al., 2019). To create the 5 tasks, the MNIST digits are split into 5 sets of 2 classes each and tasks are trained one after another with no replay of previous inputs. The network architecture was a simple MLP with one hidden layer of 1000 neurons, as in Bricken et al. (2023). I trained each network for only 1 epoch to save compute.

The goal of the experiment was to test the effectiveness of various sparse activation functions with standard optimizers tuned for continual learning. For each (activation function, optimizer) pair tested, I ran a hyperparameter sweep to find the best final mean accuracy. Before running the sweep, I manually estimated the best hyperparameters, listed in A.6. Shown in table 1 are the best result for the main activation functions of the study. For full table of results see A.5.

All of sparse activation functions performed better than all of the non-sparse functions. Hard ASH was the best in almost every optimizer setup, followed by Top-K and ASH. Adagrad performed the best out of the optimizers tested, followed by RMSprop(Tieleman & Hinton, 2012) and AdamKingma & Ba (2014). I also ran SGD and SGDM for comparison, but both of these had lower accuracy than the optimizers with adaptive, per parameter, learning rates.

Table 1: Activation functions with best performance across tested optimizers. Average of 5 runs and 95% C.I. EWC(Kirkpatrick et al., 2017), FlyModel(Shen et al., 2021) and SDMLP+EWC baseline results are from Bricken et al. (2023) and also use an MLP with a single 1000 neuron hidden layer.

Activation / Method	**Epochs**	**Mean accuracy**	**Best optimizer**
ASH	1	76.4% (±1.4%)	Adagrad
Hard ASH	1	**78.3%** (±1.4%)	Adagrad
Top-K	1	76.0% (±1.6%)	Adagrad
ReLU	1	49.2% (±7.9%)	Adam
EWC	500	61%	SGD
SDMLP	500	69%	SGD
SDMLP+EWC	500	**83%**	SGDM
FlyModel	1	77%	Association rule learning

4 Conclusions

This study challenges the conventional approaches to continual learning by demonstrating that the we can get decent results in Split-MNIST even without any continual learning algorithms or task-related information. Hard ASH performed very well in my experiments. I suggest trying it as a faster to compute alternative to Top-K, that might also boost accuracy.

Choosing the best optimizer is a hard problem given the amount of options to choose from and the amount of hyperparameter tuning required for good performance. Schmidt et al. (2021) alone listed over 100 known algorithms, each of which can optionally be paired with various learning rate schedules. For now, I recommend Adagrad as a relatively easy to tune default for continual learning. Appendices A.8 and A.9 provide more insight into optimizer performance.

5 URM Statement

The author acknowledges that at the author of this work meets the URM criteria of ICLR 2024 Tiny Papers Track.

References

Subutai Ahmad and Luiz Scheinkman. How can we be so dense? the benefits of using highly sparse representations, 2019.

Rahaf Aljundi, Francesca Babiloni, Mohamed Elhoseiny, Marcus Rohrbach, and Tinne Tuytelaars. Memory aware synapses: Learning what (not) to forget, 2018.

Jordan T. Ash and Ryan P. Adams. On warm-starting neural network training, 2020.

Trenton Bricken, Xander Davies, Deepak Singh, Dmitry Krotov, and Gabriel Kreiman. Sparse distributed memory is a continual learner, 2023.

Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. Binaryconnect: Training deep neural networks with binary weights during propagations, 2016.

Shibhansh Dohare, Richard S. Sutton, and A. Rupam Mahmood. Continual backprop: Stochastic gradient descent with persistent randomness, 2022.

John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of machine learning research*, 12(7), 2011.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification, 2015.

Yen-Chang Hsu, Yen-Cheng Liu, Anita Ramasamy, and Zsolt Kira. Re-evaluating continual learning scenarios: A categorization and case for strong baselines, 2019.

Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014. URL `https://api.semanticscholar.org/CorpusID:6628106`.

James Kirkpatrick, Razvan Pascanu, Neil Rabinowitz, Joel Veness, Guillaume Desjardins, Andrei A. Rusu, Kieran Milan, John Quan, Tiago Ramalho, Agnieszka Grabska-Barwinska, Demis Hassabis, Claudia Clopath, Dharshan Kumaran, and Raia Hadsell. Overcoming catastrophic forgetting in neural networks. *Proceedings of the National Academy of Sciences*, 114(13): 3521–3526, March 2017. ISSN 1091-6490. doi: 10.1073/pnas.1611835114. URL `http://dx.doi.org/10.1073/pnas.1611835114`.

Qingfeng Lan and A. Rupam Mahmood. Elephant neural networks: Born to be a continual learner, 2023.

Kyungsu Lee, Jaeseung Yang, Haeyun Lee, and Jae Youn Hwang. Stochastic adaptive activation function, 2022.

Alireza Makhzani and Brendan Frey. k-sparse autoencoders, 2014.

Jacob Menick, Erich Elsen, Utku Evci, Simon Osindero, Karen Simonyan, and Alex Graves. A practical sparse approximation for real time recurrent learning, 2020.

Martial Mermillod, Aurélia Bugaiska, and Patrick BONIN. The stability-plasticity dilemma: investigating the continuum from catastrophic forgetting to age-limited learning effects. *Frontiers in Psychology*, 4, 2013. ISSN 1664-1078. doi: 10.3389/fpsyg.2013.00504. URL `https://www.frontiersin.org/articles/10.3389/fpsyg.2013.00504`.

Tim Salimans and Diederik P. Kingma. Weight normalization: A simple reparameterization to accelerate training of deep neural networks, 2016.

Robin M. Schmidt, Frank Schneider, and Philipp Hennig. Descending through a crowded valley - benchmarking deep learning optimizers, 2021.

Jonathan Schwarz, Jelena Luketina, Wojciech M. Czarnecki, Agnieszka Grabska-Barwinska, Yee Whye Teh, Razvan Pascanu, and Raia Hadsell. Progress & compress: A scalable framework for continual learning, 2018.

Yang Shen, Sanjoy Dasgupta, and Saket Navlakha. Algorithmic insights on continual learning from fruit flies, 2021.

Rupesh K Srivastava, Jonathan Masci, Sohrob Kazerounian, Faustino Gomez, and Jürgen Schmidhuber. Compete to compute. In C.J. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K.Q. Weinberger (eds.), *Advances in Neural Information Processing Systems*, volume 26. Curran Associates, Inc., 2013. URL `https://proceedings.neurips.cc/paper_files/paper/2013/file/8f1d43620bc6bb580df6e80b0dc05c48-Paper.pdf`.

Tijmen Tieleman and Geoffrey Hinton. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude, 2012.

Chang Xiao, Peilin Zhong, and Changxi Zheng. Enhancing adversarial defense by k-winners-take-all, 2019.

A Appendix

A.1 Reproducibility

The code to reproduce the experiments in this paper is publicly available here:

`https://github.com/LesserScholar/hard-ash`

Care has been taken to make sure the results are reproducible, including consistent use of JAX keys and deterministic shuffling of data.

A.2 Hard ASH formula

The formula I used for Hard ASH is:

$$HardSigmoid(x) = \frac{clip(x+3,0,6)}{6}$$
$$HardASH(x_i) = clip(x_i, 0, x_{max}) \cdot HardSigmoid(\alpha \cdot (x_i - \mu_X - z_k \cdot \sigma_X))$$

Where x_{max} is a hyperparameter that I always set to 2. The formula for hard sigmoid is the one used in in JAX (jax.nn.hard_sigmoid).

Together the clip, the hard sigmoid and a high enough value for α cause most of the activations to be saturated (at either 0 or 2) and reduce amount of flowing gradients. Intuitively, this means that for each training example, I only update the incoming and outgoing weights for the activations where the network is unsure if that particular x_i should be on or off.

A.3 Network Initialization

In all of my tests I used the standard Kaiming initialization (He et al., 2015) for both layers of the MLP. It is likely that there are more efficient ways to initialize a Hard ASH network that account for the sparsity in the activations, but those explorations were not included in this study.

A.4 Weight normalization

I used weight normalization (Salimans & Kingma, 2016) with a fixed g of 1, only on the first layer of the MLP. In preliminary testing, weight normalization on the first layer consistently increased performance of multiple methods by 1 to 2 percentage points. Weight normalization on the second layer was, in preliminary testing, either net neutral or slightly negative.

A.5 Full results

Tables 2 and 3 show how all the methods and method pairs stacked against one another.

It is noteworthy that all the sparse activation functions kept a decent chunk of their performance even with basic SGD.

In the full results I tried two different versions of Top-K, subtract and mask. The difference between the two methods is that in Top-K subtract, the *k*-th highest value is subtracted from the activations before masking. In the main text only Top-K subtract is used, since it performs better. Bricken et al. (2023) also found Top-K subtract to perform better than Top-K mask.

I also tested LWTA(Srivastava et al., 2013; Xiao et al., 2019) which has been suggested as a faster to compute alternative to Top-K, but found it's performance to be worse than Top-K subtract or ASH based functions.

Table 2: Best results for each optimizer. Average of 5 runs and 95% C.I.

Optimizer	**Mean Accuracy**	**Best Activation Function**
Adagrad	**78.3%** (±1.4%)	Hard ASH
RMSprop	77.7% (±1.8%)	Hard ASH
Adam	71.6% (±1.6%)	Hard ASH
SGDM	66.3% (±3.4%)	Top-K Subtract
SGD	52.9% (±5.3%)	Hard ASH

Table 3: Full results for all tested activation functions with all tested optimizers. Average of 5 runs and 95% C.I. Poorly performing combinations were terminated early to save compute and thus have larger error bounds.

Activation	Adagrad	RMSprop	Adam	SGDM	SGD
ASH	76.4 ± 1.4	75.7 ± 1.2	69.5 ± 2.0	65.0 ± 0.4	52.4 ± 8.6
Hard ASH	78.3 ± 1.4	77.7 ± 1.8	71.6 ± 1.6	65.1 ± 1.6	52.9 ± 7.6
Top-K subtract	76.0 ± 1.6	75.0 ± 2.9	71.5 ± 1.4	66.3 ± 3.5	51.5 ± 10.4
Top-K mask	65.0 ± 4.6	69.7 ± 0.8	67.9 ± 2.4	62.9 ± 3.9	44.1 ± 15.3
LWTA	67.2 ± 2.6	67.1 ± 2.0	64.9 ± 2.2	61.3 ± 4.1	39.9 ± 14.4
ReLU	43.8 ± 10.7	39.7 ± 11.7	49.2 ± 9.7	35.7 ± 2.2	19.8 ± 4.0
SwisH	46.2 ± 9.7	41.4 ± 10.1	49.2 ± 9.7	51.9 ± 2.8	26.5 ± 8.3
Sigmoid	52.4 ± 10.3	44.2 ± 9.7	35.3 ± 8.2	20.8 ± 5.0	15.6 ± 1.0
Hard Sigmoid	56.5 ± 1.2	48.1 ± 11.6	32.0 ± 9.2	19.4 ± 1.4	14.7 ± 0.8

A.6 Hyperparameters

Tables 4 and 5 list the hyperparameters for each method used in the study.

The optimizer parameters are heavily tuned towards better performance in Split-MNIST. For momentum optimizers the momentum values were set higher than usual. For RMSprop the decay is set very high. For Adagrad the initial fill value is set unusually low which makes it's behavior similar to RMSprop in the beginning of the training, i.e. the learning rates are very high at the start of the training.

For Top-K the best k values were 64 and 96, corresponding to sparsity between 94% and 90%. For LWTA the best amounts of groups were 25 and 50, 97.5% and 95% sparse, respectively. Lee et al. (2022) shows how to calculate density of ASH based on value of z_k. I found that a z_k value of 2.0-2.5 yielded the best results, which corresponds roughly to 97-99% sparsity.

Table 4: Common hyperparameters

Hyperparameter	Values
Batch size	64
First layer weight norm	True
Second layer weight norm	False
Hidden size	1000
Gradient clip	0.01

Table 5: Method Specific Hyperparameters

Method	Hyperparameter	Values
Activations		
Ash	α	3.0, 4.0
	Z_k	2.2, 2.3, 2.4
	Hard ASH x_{max}	2.0
Top-K	k	32, 64, 96, 128, 256
LWTA	groups	25, 50, 100
Optimizers		
RMSprop	decay	0.998, 0.999, 0.9991, 0.9992, 0.9993
	learning rate	4e-6, 5e-6, 5.5e-6, 6e-6, 8e-6
Adam	β_1	0.9, 0.95, 0.98, 0.99
	β_2	0.999, 0.9995
	learning rate	8e-6, 1e-5, 1.5e-5
Adagrad	learning rate	1e-4, 2e-4, 3e-4
	initial value	1e-6
SGD	learning rate	3e-4, 4e-4, 5e-4
SGDM	momentum	0.99, 0.992, 0.994, 0.996
	learning rate	8e-6, 1e-5, 1.5e-5

A.7 Performance without task splits

The continual learning optimized hyperparameters used in the main experiment sweep have been tuned only for Split-MNIST task. Using such high adaptive learning rate and momentum parameters is very unusual. To assess how much performance is lost using these settings in typical MNIST without the task splits, I ran through the same sweep of activation functions and optimizers but training all classes simultaneously, i.e. with i.i.d. data.

Table 6 shows the results when trained on whole MNIST at once (i.e. with i.i.d. dataset). First sweep was with the same hyperparameters used in the main experiment, tuned for continual learning, and second with the optimizer hyperparameters tuned to i.i.d data. In the second case, the much lower momentum and higher learning rates, allow the model to reach much better accuracy in 1 epoch.

Table 6: Comparison of split task continual learning scenario, i.i.d. with continual learning optimizer and i.i.d with regular optimizer. Average of 5 runs and 95% C.I.

Method	Epochs	Mean accuracy
Hard ASH /w task splits and Split-MNIST optimizer	1	78.3% (±1.4%)
ReLU i.i.d. dataset and Split-MNIST optimizer	1	91.3% (±2.5%)
ReLU i.i.d. dataset and normal optimizer	1	96.9% (±3.3%)

A.8 Adam and bias correction

In the main text I focused on evaluating existing and well studied optimizers without modifications. One of the more surprising results was how much worse Adam performed when compared to very similar RMSprop and Adagrad algorithms. After testing I found that I can boost the Adam performance almost to RMSprop level simply by removing bias correction from the algorithm.

Bias correction in Adam adjusts the first and second moment estimates to account for their initialization at zero. This makes the moment estimations more accurate (i.e. unbiased) during the early optimization steps.(Kingma & Ba, 2014) Unlike Adam, standard implementations of RMSprop do not use bias correction. RMSprop and biased Adam performing significantly better than standard Adam suggests to us that the initial very high learning rates given by the biased exponential averaging are important for the final accuracy of the network.

Table 7: Testing Adam performance with bias correction removed. Average of 5 runs and 95% C.I.

Optimizer	**Mean Accuracy**	**Activation Function**
Adam with bias correction	71.6% (±1.5%)	Hard ASH
Adam without bias correction	76.7% (±1.5%)	Hard ASH

A.9 Learning rate schedules

In my experiment adaptive learning rate methods did very well, but I did not test learning rate schedules. Continual learning experiments are often done with constant learning rates, to maintain plasticity (for more on plasticity see A.10). But the success of Adagrad, RMSprop and biased Adam suggests that the initial high learning rate is important. Therefore I also tested SGD with exponential decay schedule and found it to be fairly effective. I got accuracy of **72.4%** (±3.2%), combining Hard ASH with exponentially decaying learning rate with decay of 0.7 every 200 steps, with starting learning rate of $3.35e - 3$. The parameters for exponentially decaying learning rate are fairly sensitive and much harder to tune than something like Adagrad with a constant learning rate.

A.10 Plasticity experiment on permuted MNIST

Of primary concern in continual learning is the so called stability-plasticity dilemma Mermillod et al. (2013), which highlights the tension between remembering past tasks and learning to perform on new tasks. As an extreme example, it would be easy to build a learning system that perfectly remembers the early tasks and then stops learning, simply by setting the learning rate close to 0 after certain amount of training steps. But this would be against the spirit of continual learning since the network would completely stop absorbing new knowledge. So the end goal of continual learning is to build a learning system where the performance on old tasks is stable and it keeps learning new tasks easily.

To assess the effect that sparsity has on plasticity, I ran another smaller experiment on the permuted MNIST dataset (Kirkpatrick et al., 2017). In permuted MNIST, each subsequent task in a sequence of tasks is created by applying a fixed permutation to the pixels of the original MNIST images, resulting in different, but structurally similar, tasks. Since you can keep applying new permutations at will, you can keep training on new tasks almost forever.

To test the effectiveness of sparsity on plasticity, I ran this experiment on Hard ASH with different hyperparameters. The hypothesis tested was that changing the amount of sparsity (modifying z_k) or gradient sparsity (modifying α, steeper alpha curve results in more sparsity in the gradients) would have an effect on the plasticity of the network.

The experiment was ran for 100 epochs, changing to a new task after every epoch for a total of 100 tasks. Adagrad was chosen as the optimizer since it was the overall best performer in the main study. Network architecture is the same as in the main experiment, an MLP with a single 1000 neuron hidden layer.

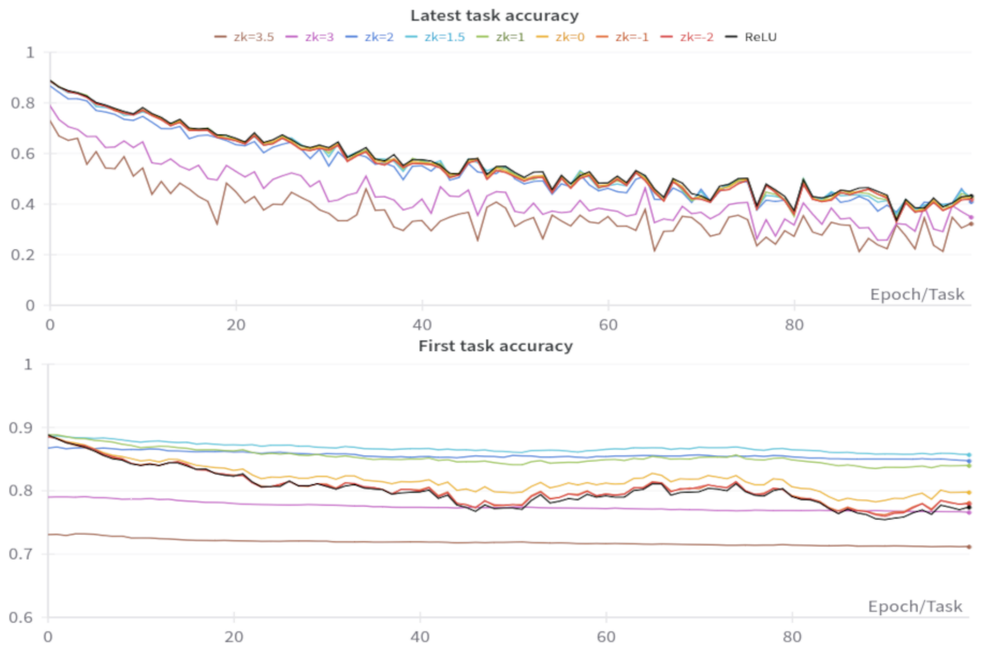

Figure 2: Latest task and first task validation accuracies when varying z_k

Figures 2 and 3 show the accuracy on the latest task right after it's training is complete (plasticity) and performance on the very first task through the whole training (stability).

In figure 2, I show the effect of varying z_k, i.e. the amount of sparsity in the hidden layer representation. Latest task accuracy graph shows that varying the amount of sparsity has negligible effect on plasticity. When $z_k < 3$, the Hard ASH MLP has pretty much same plasticity as baseline ReLU MLP. When $z_k \geq 3$, the performance is already worse during the first epoch, but it doesn't appear to help much with plasticity.

In the first task accuracy graph we see that the networks ability to retain accuracy on the first task increases with the amount of sparsity. When $z_k < 1$, the network performs similarly to the baseline ReLU, but when $z_k > 1$, the performance on the first task falls only slightly during the training of the 99 subsequent tasks.
In figure 3, I show the effect of varying α or the steepness of the activation slope, i.e. the amount of sparsity in the gradients. For this experiment, z_k was held constant at 1.5. Latest task accuracy graph shows much of the same as figure 2. Plasticity is the same regardless of the settings and pretty much matches the baseline ReLU. The stability, or the ability to retain accuracy on the very first task, increases with α. When α gets very high (e.g. 8, as shown in the graph) there is a noticeable performance degradation already at the beginning of the training, but no significant benefits to plasticity.

Thus my conclusion is that representation sparsity alone is not enough to solve plasticity in continual learning and something extra is required. Instead I can say that sparsity helps with stability without significant penalty to plasticity. For a more complete continual learner, that overcomes both sides of stability-plasticity dilemma, we could try to combine sparse representations with a technique that increases plasticity, such as Continual backprop (Dohare et al., 2022), Shrink and perturb (Ash & Adams, 2020) or progressive magnitude based pruning with model expansion like in Menick et al. (2020).

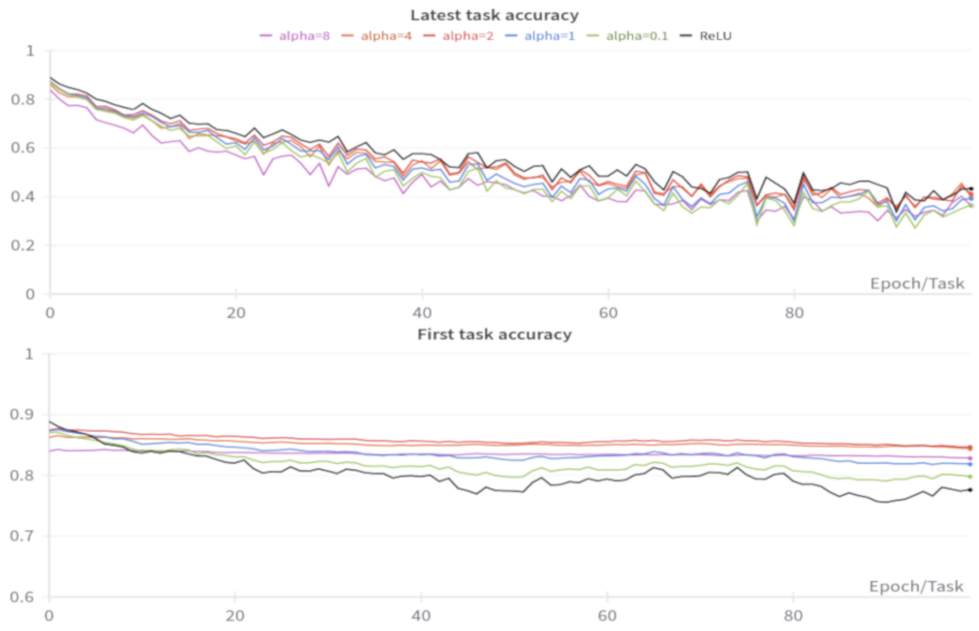

Figure 3: Latest task and first task validation accuracies when varying α