PERSISTENT INTERNAL STATE HELPS MAINTAIN LEARNING PLASTICITY

Nabil Iqbal
Deptartment of Mathematical Sciences, Durham University
AMLab, University of Amsterdam
nabil.iqbal@durham.ac.uk

ABSTRACT

A failure mode associated with continual learning is a loss of *plasticity*, in that a neural network that is repeatedly trained eventually ceases to be able to adapt to changing stimuli. In this work we show that plasticity loss can be mitigated by maintaining a new internal state – persistent from batch to batch – in the network. In training, the weights of the network are updated by conventional gradient descent, and this extra internal state is updated by a learned rule. This internal state provides an extra mechanism for the network to retain a memory of previous problems, in addition to the conventional weights. We empirically demonstrate the benefits in simple benchmarks.

1 MOTIVATION

In this work we will study the phenomenon of *plasticity loss* in continual learning, see e.g. Dohare et al. (2024). Plasticity loss refers to the phenomenon by which repeatedly training a neural network reduces its ability to train on a new dataset. In this work we demonstrate a simple mechanism for mitigating this plasticity loss which does not require explicit construction of new learning rules. Our mechanism involves the addition of a persistent and self-adjusted internal state to the network, in addition to the weights.

1.1 PREVIOUS WORK

Plasticity loss in deep learning was demonstrated empirically in Dohare et al. (2024) and has been studied in a number of works, including Lyle et al. (2025; 2023). The problem was tackled with an explicit separation of timescales in Lee et al. (2024). There is extensive study within both deep learning and neuroscience of synaptic memories and modified learning rules, including Zenke et al. (2017); Benna & Fusi (2016); Kaplanis et al. (2018). Our mechanism is somewhat simpler, in that the internal state is global rather than attached to each neuron, and the network learns itself what to do and maintains different timescales if necessary. There is some conceptual overlap with the goals of meta-learning (see e.g. Javed & White (2019) and in particular Beaulieu et al. (2020)), though as far as we can tell our precise training mechanism – which does not involve meta-learning – and the application to plasticity loss are novel.

2 MECHANISM

We will consider a neural network f_θ which learns a function from an input space $\mathbb{R}^d$ to an output space $\mathbb{R}^{d'}$, i.e.

$$y = f_\theta(x) \qquad y \in \mathbb{R}^{d'}, x \in \mathbb{R}^d \tag{1}$$

This network is specified by a set of weights θ. We will consider a continual learning problem, in which we have T tasks and the network is trained on each of them in sequence. We denote each training step by t.

2.1 BACKGROUND

In a conventional setup, the weights at step t are updated as:

$$\theta_{t+1} = \theta_t - \nabla_\theta \mathcal{L}(\theta_t, x_t) \tag{2}$$

where $\mathcal{L}$ is the train loss and $x_t \in \mathbb{R}^{d \times \text{batch size}}$ denotes a batch of training data at step t. (In some cases this batch is taken to have only a single data point). As t progresses, we eventually we exhaust a given task and move on to a new one. It is well-known that this procedure generally results in a gradual reduction of network performance on new tasks: attempting to re-train a *trained* network results in inferior performance, as can be seen in the baseline plots in Figure 2.

It is interesting to take a step back and ask how (and if) information on problem solving is transferred from task to task as t continues. The network does not really have any explicit tools to do this. Presumably some information is stored in the weights, but the *network* does not decide how the weights are updated: instead the SGD rule (2) blindly updates them in a manner to reduce the instantaneous loss. One might imagine that if the network had more control on how information was stored it would transition from task to task more effectively.

2.2 Persistent internal state

We thus consider the following setup: in addition to the regular weights θ, we also add to the *network* a persistent internal state $h \in \mathbb{R}^{d_{\text{int}}}$. The network itself is now characterized by (θ_t, h_t). The output of the network at each step t is now a learnable function which depends both on the training data x_t and the value of the internal state h_t:

$$y_t = f_{\theta_t}(x_t, h_t) \tag{3}$$

We now consider the following training procedure: at each training step, we update the internal state h_t through a *learned rule* g_θ:

$$h_{t+1} = g_{\theta_t}(x_t, h_t) \tag{4}$$

after which we update the weights by standard gradient descent:

$$\theta_{t+1} = \theta_t - \nabla_\theta \mathcal{L}(\theta_t, x_t, h_t) \tag{5}$$

(Note that for notational simplicity we use the notation θ_t to refer to the all of the weights; in reality a subset of them defines f_{θ_t} and a distinct subset defines g_{θ_t}). Thus the internal state h_t is persistent from batch to batch and acts as a very primitive sort of training memory. h_t somewhat blurs the line between a weight and an internal state. To our knowledge this architecture – though very simple – is novel in this context. In the rest of this work we describe how this affects performance in continual learning. Perhaps somewhat surprisingly, it leads to concrete benefits in plasticity.

3 Experiment

We study the Permuted-MNIST benchmark (Goodfellow et al., 2013; Zenke et al., 2017), along with its Fashion-MNIST incarnation. This benchmark consists of a long sequence of T tasks. To construct a new task we choose a random permutation which acts on the flattened vector of input pixels, and then act with the same permutation on all of the images in the dataset, as in Figure 1. We continuously present the network with a batch of input images, performing the update (4) and (5) after each batch. One task consists of a single epoch where the network sees each (permuted) image once. The network is not explicitly told of the task boundaries. We use fully connected MLPs (as the permutation destroys all notion of spatial locality), resulting in a sequence of tasks which all have the same difficulty.

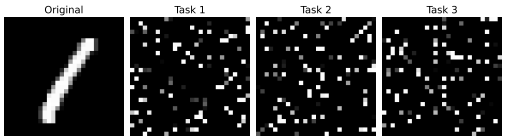


Figure 1: Example of tasks in Permuted MNIST; the pixels of the dataset are acted on by a constant permutation which is randomly chosen for each task.

3.1 Implementation

We briefly describe some details on the implementation. In practice both the output f_θ in (3) and the memory update g_θ in (4) are implemented as MLPs. Note that g_{θ_t} in (4) acts on a *batch* of data, and thus a choice must be made about how to aggregate this into a single h_{t+1}: any permutation-invariant operation acting on the batch axis would work, and we choose to take the mean before

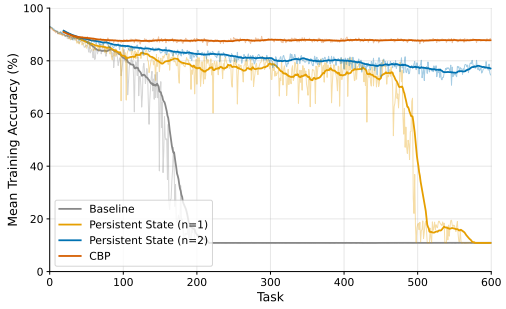
acting with a final linear layer. Furthermore, as input we simply concatenate the h_t with each x_t in the batch.

The situation described by (5) is not quite the same as regular backpropagation, nor backpropagation through time as for a recurrent neural network. In particular, the computational graph associated with h_t can grow arbitrarily long. We truncate it after n steps, where n is a hyperparameter we pick. Furthermore, the weights θ_t are continuously updated *in place*, meaning that for $n > 1$ it is not possible to properly take a gradient, as the gradient would depend on a set of values θ_t that were used in the past and have subsequently been changed. We simply ignore this fact and use the instantaneous values of θ_t for each update, in keeping with the philosophy of continual learning. This introduces a small error in the gradient updates[1]

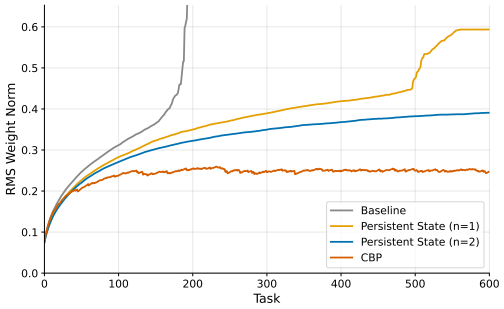
3.2 Results

Here we present a preliminary investigation. We work with a batch size of 8 and a learning rate of 0.05. Our networks are fully connected MLPs with ReLU activations and three layers all of width 100. We vary the dimension d_{int} of the internal state h_t. We track the mean accuracy across all batches for a given task as well as the accuracy on a held-out test set at the end of each task. Further details can be found in Appendix A.1.

We benchmark against (a) a baseline with no mitigations, and (b) continual backpropagation (CBP) – an algorithm which explicitly re-initializes dead neurons – as in Dohare et al. (2024; 2021). Results are for a sequence of 600 tasks are shown in Figure 2 and 3. All results below are for one seed. We find qualitatively similar results for different seeds, though the precise task at which the collapse occurs can differ.



(a) Mean training accuracy, averaged across all online batches. Bold line is sliding average of width 20.



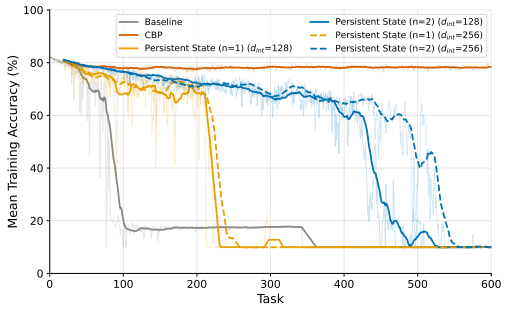
(b) Weight norm as a function of task.

Figure 2: Results for MNIST. The persistent state for $n = 2$ maintains plasticity over the length of the experiment; with $n = 1$ it survives far longer than the baseline, though it eventually collapses. The dimension of the internal state $d_{\text{int}} = 128$. Bold line is a sliding average with width 20 tasks.

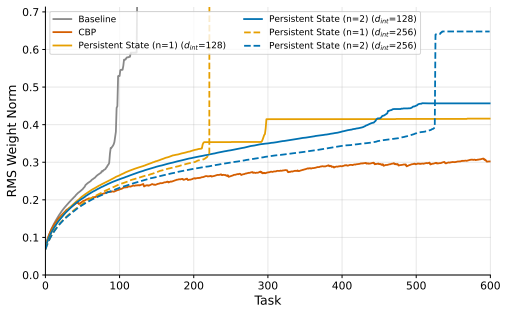
We see that at the start all methods perform well on the datasets. However performance begins to degrade as we progress through the tasks. These settings (with a relatively high learning rate and narrow network) were deliberately chosen to be very aggressive, and for the baseline with no mitigation measures they cause a rapid loss of plasticity resulting in a complete failure of trainability by around task $\sim$ 100 - 200.

In contrast, our method with a persistent state performs much better; the gradual degradation is significantly slower. As we vary the hyperparameter controlling the number of steps n before detaching h_t, performance increases from $n = 1$ to $n = 2$ for both datasets. For MNIST the collapse does not occur for $n = 2$ over the 600 tasks that we study. For the harder task of Fashion-MNIST we push back the collapse but cannot stop it completely; we also see a small increase in performance by increasing the dimension of the internal state. In Figure 4 we investigate increasing this to $n = 4$; curiously it seems that increasing n further than 2 is counterproductive.

[1]This requires overriding the usual PyTorch context manager, which normally does not allow a calculation with such an error, whether or not it is small.



(a) Mean training accuracy, averaged across all online batches. Bold line is sliding average of width 20.



(b) Weight norm as a function of task

Figure 3: Results for Fashion-MNIST. Increasing the dimension of the internal state helps performance. The best value of n appears to be $n = 2$. We vary the dimension of the internal state $d_{\text{int}} \in \{128, 256\}$.

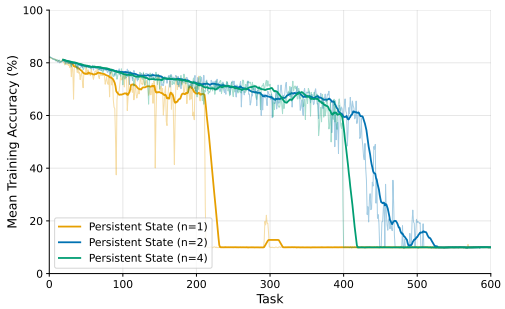


Figure 4: Comparison of different values of n on Fashion-MNIST with $d_{\text{int}} = 256$: note that performance does not increase montonically with n, and $n = 2$ is the best.

To guarantee that the effect arises from the dynamics of the internal state and not the small change in architecture arising from its inclusion, in Figure 6 we also consider ablations in which we keep the internal state in the architecture but either set it to zero or randomize its elements (while preserving its norm) at each timestep. These both perform essentially identically to the baseline.

3.3 Discussion

In all of these methods the best performance comes *not* from our work but from the continual backpropagation algorithm of Dohare et al. (2024), which largely solves plasticity loss entirely. However, that algorithm is hand-designed to solve this particular problem; in contrast, our approach is entirely self-organizing and learned by the network itself. It would very interesting to understand the mechanism by which this persistent state staves off the collapse.

To that end, we consider tracking the $L2$ norms of the weights (normalized by their total number, which differs slightly from model to model), shown in the right panel of Figures2 and 3. The growth of these norms is associated with plasticity loss (Lyle et al., 2025). We see that indeed the best performing models have the slowest growth of the weights.

At present we do not have a mechanistic understanding of this for our persistent state architecture. A reasonable guess is that the internal state preserves some memory of the last task which allows the weights to generalize to the new task without requiring them to change as much[2]. To investigate this we considered *alternating* the tasks of Fashion-MNIST and MNIST in sequence. If the hypothesis above was correct then we would expect dramatically reduced performance of the internal state mechanism, since a memory of the last task is no longer useful for the new one at all. However we instead find performance quite comparable to the case without alternation (see Figure 7). Thus at present we feel the internal state acts instead as a kind of regulator without significant information content, though this remains to be explored.

This is the simplest possible implementation of the idea, and we have not attempted any serious engineering. We thus find it plausible that the performance of this mechanism could be improved until it is competitive with hand-designed algorithms such as continual backpropagation. One concrete approach is to try more sophisticated models for the internal state (e.g. perhaps a more explicit form of memory such as Krotov & Hopfield (2016), Graves et al. (2014), or Behrouz et al. (2024)). Optimistically, we hope that the mechanism that we have displayed here – simple though it is – might lead to new ideas and ways to think about continual learning.

[2]For example, for the Permuted-MNIST benchmarks, an appropriately chosen permutation on the first layer – while holding the other layers fixed – is enough to solve the new task completely. One might imagine that the persistent internal state provides a mechanism for this to take place.

ACKNOWLEDGMENTS

NI is supported in part by the STFC under grant number ST/T000708/1.

REFERENCES

Shawn Beaulieu, Lapo Frati, Thomas Miconi, Joel Lehman, Kenneth O Stanley, Jeff Clune, and Nick Cheney. Learning to continually learn. *arXiv preprint arXiv:2002.09571*, 2020.

Ali Behrouz, Peilin Zhong, and Vahab Mirrokni. Titans: Learning to memorize at test time. *arXiv preprint arXiv:2501.00663*, 2024.

Marcus K Benna and Stefano Fusi. Computational principles of synaptic memory consolidation. *Nature neuroscience*, 19(12):1697–1706, 2016.

Shibhansh Dohare, Richard S Sutton, and A Rupam Mahmood. Continual backprop: Stochastic gradient descent with persistent randomness. *arXiv preprint arXiv:2108.06325*, 2021.

Shibhansh Dohare, J Fernando Hernandez-Garcia, Qingfeng Lan, Parash Rahman, A Rupam Mahmood, and Richard S Sutton. Loss of plasticity in deep continual learning. *Nature*, 632(8026): 768–774, 2024.

Ian J Goodfellow, Mehdi Mirza, Da Xiao, Aaron Courville, and Yoshua Bengio. An empirical investigation of catastrophic forgetting in gradient-based neural networks. *arXiv preprint arXiv:1312.6211*, 2013.

Alex Graves, Greg Wayne, and Ivo Danihelka. Neural turing machines. *arXiv preprint arXiv:1410.5401*, 2014.

Khurram Javed and Martha White. Meta-learning representations for continual learning. *Advances in neural information processing systems*, 32, 2019.

Christos Kaplanis, Murray Shanahan, and Claudia Clopath. Continual reinforcement learning with complex synapses. In *International Conference on Machine Learning*, pp. 2497–2506. PMLR, 2018.

Dmitry Krotov and John J Hopfield. Dense associative memory for pattern recognition. In *Proceedings of the 30th International Conference on Neural Information Processing Systems*, pp. 1180–1188, 2016.

Hojoon Lee, Hyeonseo Cho, Hyunseung Kim, Donghu Kim, Dugki Min, Jaegul Choo, and Clare Lyle. Slow and steady wins the race: Maintaining plasticity with hare and tortoise networks. In *Forty-first International Conference on Machine Learning*, 2024. URL `https://openreview.net/forum?id=VF177x7Syw`.

Clare Lyle, Zeyu Zheng, Evgenii Nikishin, Bernardo Avila Pires, Razvan Pascanu, and Will Dabney. Understanding plasticity in neural networks. In *International Conference on Machine Learning*, pp. 23190–23211. PMLR, 2023.

Clare Lyle, Zeyu Zheng, Khimya Khetarpal, Hado van Hasselt, Razvan Pascanu, James Martens, and Will Dabney. Disentangling the causes of plasticity loss in neural networks. In *Conference on Lifelong Learning Agents*, pp. 750–783. PMLR, 2025.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerner, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*, 2019.

Friedemann Zenke, Ben Poole, and Surya Ganguli. Continual learning through synaptic intelligence. In *International conference on machine learning*, pp. 3987–3995. PMLR, 2017.

A Appendix

A.1 Architectural details

We briefly describe some details of the architecture. The baseline model is simply an MLP with ReLU activations; input is a d-dimensional vector, and we have three hidden layers of width 100 before an output of d' classification logits. For MNIST we have $d = 784, d' = 10$.

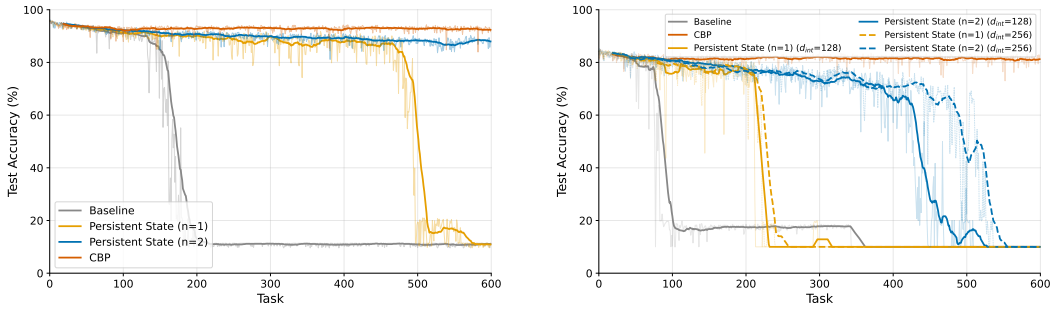
For the persistent internal state model, the input is a $d + d_{\text{int}}$ dimensional vector which is the concatenation of the d-dimensional data vector and the d_{int} internal state vector h_t. We then have three hidden layers of width 100. At the end there are two heads; the *task head* is a linear layer that is $100 \times d'$, acting on the penultimate layer of activations and producing a d'-dimensional output of classification logits. We also have a *state head*; this is a linear $100 \times d_{\text{int}}$ layer which acts on the *average* over the batch dimension of the output from the network, to output a single state update as in (4) per batch.

During training we detach the internal state after n steps, where n is a number that we can pick. In our trials of $n \in \{1, 2, 4\}$, we found $n = 2$ to be the best, though $n = 1$ is still significantly better than the baseline. This is interesting, as in the case of $n = 1$ the output from the state head is detached before it ever enters the loss function. Thus the SGD updates in (5) do not ever update the weights in the state head, which remains identical to its value at initialization. One can imagine that this means that a random projection of the penultimate layer is stored in the internal state; nevertheless this still seems to be of value to the network. For $n = 2$ and above the network can now optimize what it chooses to store in the internal state.

Our models are implemented in PyTorch (Paszke et al., 2019). We ran two seeds for the experiments shown above and found qualitatively similar results for both, though the precise location of the collapse can vary.

A.2 Further experimental results

We present a few further experimental results. The parameters are as described above, and the significance of the experiments is described in the main text.



(a) MNIST

(b) Fashion MNIST.

Figure 5: The test accuracy (rather than the mean online accuracy) on a held-out test set at the end of the training on each task, for the same runs as shown in Figure 2 and 3.

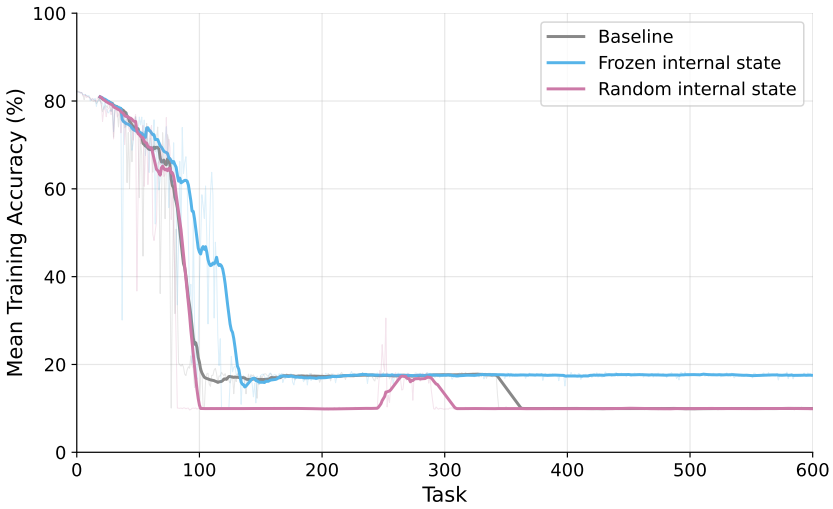


Figure 6: Ablation where we freeze the internal state to 0 or set it randomly at each timestep. We see that these both collapse essentially just as fast as the baseline, showing that the benefit arises from the dynamics of the internal state and not from the change in architecture needed to include it.

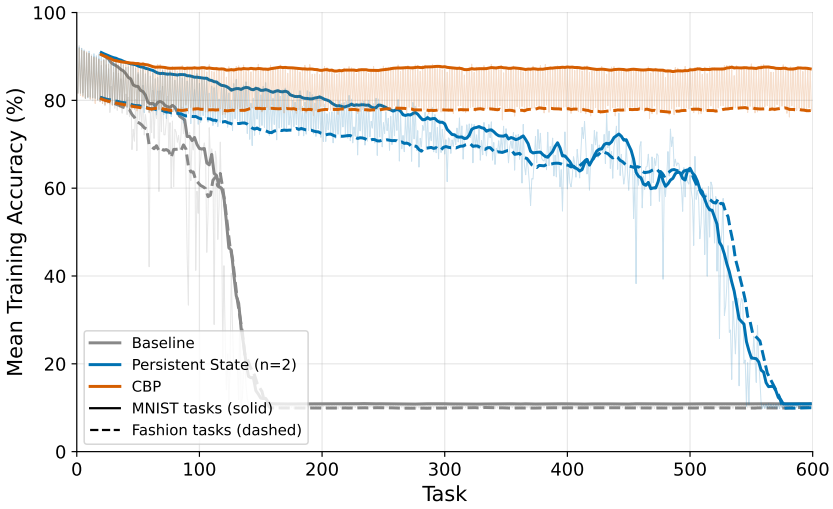


Figure 7: Alternating permuted Fashion-MNIST (average over these tasks shown as dashed) and MNIST (average over these tasks shown as solid) tasks. Note that the performance now oscillates rapidly upwards and downwards, as the difficulty of MNIST differs from that of Fashion-MNIST, but the performance of the persistent state method is roughly the same as that on only Permuted Fashion-MNIST.