Selective Reinitialization Algorithms for Preventing Plasticity Loss in Artificial Neural Networks

by

Juan Fernando Hernandez Garcia

A thesis submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

Department of Computing Science

University of Alberta

Abstract

In this dissertation, I study systems based on artificial neural networks that learn from non-stationary data. Learning from non-stationary data requires continual adaptation of the system, and is often referred to as *continual learning*. Developing systems capable of continual learning is a longstanding goal in artificial intelligence. In deep learning, the field concerned with designing and training deep neural networks, this goal remains elusive.

This dissertation addresses a fundamental challenge in continual learning: *loss of plasticity*, a phenomenon where artificial neural network systems progressively lose their ability to learn from new data. While the phenomenon has been noted several times over the last three decades, it has remained understudied until recently. The work in this dissertation constitutes the first systematic demonstrations of plasticity loss, highlighting its persistence and importance.

In this work, I provide a systematic demonstration of plasticity loss in a wide variety of deep learning systems. Across systems based on fully-connected networks, convolutional networks, residual networks, and vision transformers, plasticity degrades when learning continually. Notably, even systems employing normalization techniques, residual connections, and regularization—design choices that improve training stability—remain susceptible to plasticity loss. This evidence establishes that plasticity loss is pervasive and that deep learning systems trained with backpropagation are not suitable for continual learning.

I explore the idea of selective reinitialization to prevent plasticity loss. This idea has been used in the past for improving generalization performance in deep learning systems. However, its use for preventing plasticity loss is a recent innovation pioneered by the continual backpropagation

algorithm. The algorithm periodically sets new values for units in the network, making initialization a continuous process rather than a one-time operation. I demonstrate the effectiveness of continual backpropagation in preventing plasticity loss across a wide variety of settings. These demonstrations establish that plasticity loss, while pervasive, is not inherent to deep learning systems.

I generalize continual backpropagation through an algorithm I call *selective unit reinitialization*. This general algorithm has three key components: a utility measure that ranks units by importance, a pruning criterion that selects which units to reinitialize, and a reinitialization method that assigns new values to the selected units. Continual backpropagation involves a specific choice of utility measure, pruning criterion, and reinitialization method. However, the general algorithm is not limited to the choices used in continual backpropagation. I present a study of how different choices for the three components of selective unit reinitialization affect its effectiveness at maintaining plasticity. This study establishes selective unit reinitialization as a general approach that can be tailored to each learning system to maintain plasticity.

Finally, I propose a different approach to implementing selective reinitialization that operates at the weight level. I call the corresponding algorithm *selective weight reinitialization*. Reinitializing at the unit or weight level involves different trade-offs. Unit reinitialization minimally disrupts the network outputs, preserving stability during learning. However, the definition of a unit varies across network architectures, requiring additional engineering to make the approach effective. Weight reinitialization, in contrast, can be readily applied to any arbitrary network architecture, but can substantially affect network outputs, reducing training stability. The reduced learning stability can be remedied with L2 regularization, which stabilizes selective weight reinitialization but introduces an additional hyperparameter. Both selective unit and weight reinitialization successfully maintained plasticity across the systems tested, providing flexible approaches for different architectural and engineering constraints.

Preface

Chapters 3, 6, and 7 are based on the paper:

- Dohare, S., Hernandez-Garcia, J.F., Lan, Q. et al. Loss of plasticity in deep continual learning. *Nature* **632**, 768–774 (2024). https://doi.org/10.1038/s41586-024-07711-7.

Shibhansh Dohare, Richard Sutton, Parash Rahman, and I jointly designed the Continual ImageNet problem. Shibhansh worked on demonstrating plasticity loss and the performance of continual backpropagation in Continual ImageNet, which are results I do not include in this dissertation but extend upon. I worked on demonstrating loss of plasticity in the Incremental CIFAR-100 problem using ResNet-18, as well as showing that continual backpropagation was capable of preventing the phenomenon. Shibhansh and I analyzed those results together and have both included them in our dissertations. Shibhansh and I collaborated to demonstrate plasticity loss in Permuted MNIST across several deep learning systems, which I do not include in this dissertation but expand upon by using mini-batches instead of processing one sample at a time. Shibhansh and I worked on the literature review about the loss of plasticity and reinitialization methods, and both of us have included it in our dissertations. Lastly, Shibhansh and I worked together on measuring the correlates of plasticity loss. I include the analysis of the correlates of plasticity loss in the Incremental CIFAR-100 problem in this dissertation.

The results in Chapters 4, 5, and 7 are based on the paper

- Hernandez-Garcia, J. F., Dohare, S., Luo, J., and Sutton, R. S. (2025). Reinitializing weights vs units for maintaining plasticity in neural networks. In *4th Conference on Lifelong Learning Agents*,

which was selected for an oral presentation. All the results in the paper are included in this dissertation along with several extensions. For easy access, the code for running all the experiments in this dissertation is available at `https://github.com/JFernando4/plasticity-via-reinit`.

Dedicado a mi familia,
la que he encontrado, la que he redescubierto, y la que estoy por commenzar.

Acknowledgements

First and foremost, I would like to thank my supervisor, Rich, who has spent countless hours training me to become a better scientist and writer. I was fortunate to have Rich's clear thinking and intuition to guide me through the process of identifying problems worth studying. Yet, I am most grateful for Rich's kind approach to mentoring. The biggest obstacle during this PhD has been my recurrent belief that my ideas are not good enough, on many occasions, leading me to consider dropping out of the program. Despite my self-doubts, Rich always expressed genuine curiosity and interest in what I had to say, reassuring me that what I was accomplishing was important. It was this gentle reassurance that eventually helped me muster the confidence to defend the ideas I present in this document. For teaching me that even small contributions can have a significant impact in the face of big problems, I am forever grateful to him.

I was equally fortunate to study at the University of Alberta, where many people foster an environment that nurtures new ideas and scholarship. I am grateful to my committee members, Martha White, Rupam Mahmood, Matthew Taylor, and Irina Rish, for providing feedback on my work and pushing me to express my ideas clearly. From Michael Bowling, I learned the joy of teaching undergraduate classes. His talent for making every lecture engaging, along with his level of care for each student in the class, illustrates the kind of educator I aspire to be one day. My co-author, roommate, and close friend Shibhansh was a constant source of support and encouragement through many difficult times. I am happy I got to celebrate so many milestones with him, and I am still hopeful that some of his confidence will rub off on me someday.

There is a multitude of people in the department who have made my PhD the most rewarding time of my life. Some of them, in no particular order, are Abhishek, Leticia, Khurram, Zaheer, Graves, Roshan, Niko, Tian, Kenny, Kris, Kevin, Anna, Diego, Shadan, Athar, Farzane, Laura, Johannes, Matt, Andy P, Andy W, Aidan, Alex K, Alex L, Rory, Gábor, Edan, Esraa, Farnaz, Raksha, Suyog, Katya, Mahvash, Prabhat, Rohan, and Dylan. Many friends and family beyond the department have also been there for me, bringing a distinct kind of joy to my life. Susie and Adrian, who always make me feel at home every time I visit. Mary, Zoë, Kellie, and Randy, whom I met playing board games, but whose continued friendship over the last decade has been invaluable.

Brittany, Shane, Avideh, and Sandra for teaching me that I can also accomplish great things as a dancer. Ben, Steph, and Sam were my first close friends in Canada, and I often miss the time when we lived just minutes away from each other. Margarita has become a second mother to me, always welcoming me with a warm meal and a joyous laugh in her home. My family in Cordoba—Lelia, Mauro, and Angel—have been a source of unwavering love and support, uplifting me when I needed it most.

Finally, thank you, Amalia, for teaching me that sometimes you need a little push to achieve what you thought was impossible. For your love, support, and our life together, thank you.

Contents

List of Tables

List of Figures

List of Algorithms

Chapter 1

Introduction

This dissertation studies the ability of deep neural networks to learn from a stream of non-stationary data—a capability known as *continual learning*. Biological systems achieve this naturally, adapting continually as they encounter new experiences. However, in *deep learning*, the field concerned with studying deep neural networks, designing systems capable of continual learning remains an elusive goal.

Instead of learning continually, standard deep learning systems use a special training phase. During this special phase, the system learns from a stationary dataset in a process that involves adjusting the network's connections to produce desired outputs, which is typically achieved through backpropagation (Rumelhart et al., 1986). Once the training phase concludes, the connections of the network are frozen and the system is deployed to interact with the world without further learning. Though intermediate steps like fine-tuning or incorporating human feedback may occur, the outcome is the same, learning stops before deployment. Most advances in deep learning have emerged within this *stationary learning* framework.

Yet, deep learning systems designed under the stationary learning framework interact with a changing world. This interaction necessitates continual adaptation, which stationary learning systems lack. Current workarounds include periodic retraining from scratch or from previous checkpoints, resulting in knowledge cutoffs in systems like large language models, or loading recent information into context windows at the cost of memory. However, these are stopgap measures that do not address the fundamental limitation: standard deep learning systems are not suited for continual learning.

The fundamental challenge in designing deep learning systems suitable for continual learning is how to acquire new knowledge without forgetting useful information. This challenge is often referred to as the stability-plasticity dilemma (Mermillod et al., 2009). Plasticity refers to the ability to learn, whereas stability refers to the ability to retain previously learned information. A

lot of attention has been devoted to studying stability (McCloskey and Cohen, 1989; French, 1999), while plasticity has not received the same level of scrutiny. The disproportionately small amount of research on plasticity could lead one to wrongly conclude that plasticity is guaranteed in deep learning systems. In this dissertation, I demonstrate that continual deep learning systems lose their ability to learn over time—a phenomenon I call *loss of plasticity*.

This dissertation presents the first systematic investigation of plasticity loss in contemporary deep learning systems. Prior to this work, the most direct studies of the phenomenon were performed in the field of psychology (Ellis and Lambon Ralph, 2000; Thomas and Johnson, 2006; Ralph and Ehsan, 2006). While thorough, these studies predated deep learning, leaving open the question of whether plasticity loss manifests in deep neural network systems. Following deep learning's emergence, plasticity loss was observed on multiple occasions but not studied directly. Instead, different manifestations were investigated separately without recognizing them as instances of the same underlying phenomenon (Chaudhry et al., 2018; Ash and Adams, 2020; Igl et al., 2021; Kumar et al., 2021; Nikishin et al., 2022).

This dissertation provides direct and systematic demonstrations of plasticity loss across a wide variety of deep learning systems. I present a methodical study of plasticity loss in learning systems based on fully-connected networks, convolutional networks, residual networks, and vision transformers. Regardless of the architecture, the results show that the corresponding learning system loses plasticity over time when learning continually. This evidence establishes plasticity loss as a pervasive phenomenon in deep neural networks trained through backpropagation.

Along with the demonstrations, I also show that plasticity loss can be prevented by modifying conventional learning with backpropagation. A key insight for developing learning systems resilient to plasticity loss is to avoid operations that occur only once in a system's life span. Deep learning systems trained with backpropagation include a special operation: the initialization of the network's parameters. Initialization happens once before training and involves sampling random values from carefully designed distributions that facilitate learning. To convert initialization into a continuous process, I explore the idea of augmenting backpropagation by periodically reinitializing parts of the network—a process I call *selective reinitialization*.

Learning systems that employ selective reinitialization have a rich history. The first of those systems, introduced by Selfridge (1958), would periodically reinitialize the least useful units in a network to produce increasingly accurate predictions. Similar learning systems were later proposed, but none of them were employed for continual learning (Klopf and Gose, 1969; Holland and Reitman, 1977; Kaelbling, 1993; Mahmood and Sutton, 2013). Dohare (2020) and Rahman (2021) were the first to propose learning systems that included selective reinitialization to prevent plasticity loss in continual learning. Their work led to the proposal of the continual backpropagation algorithm by Dohare et al. (2021). The algorithm represented a significant step in demonstrating that plasticity

loss is not inherent to deep learning systems and can be prevented in fully-connected networks.

My work extends the demonstrations of continual backpropagation to a wider range of deep learning systems. Specifically, I extend the demonstrations to convolutional networks, residual networks, and vision transformers. This broader evaluation establishes continual backpropagation as an effective approach for maintaining plasticity loss across diverse deep learning architectures.

I extend my work on continual backpropagation by studying a broader class of selective reinitialization algorithms. At its core, selective reinitialization comprises three key components: a utility measure that quantifies the importance of each structure in the network, a pruning criterion that selects structures for reinitialization, and a reinitialization method that assigns new values to structures. Using this abstraction, I propose two general algorithms. *Selective unit reinitialization* operates at the unit level, generalizing continual backpropagation. *Selective weight reinitialization* operates at the level of weights, offering a novel approach to maintaining plasticity. I provide a systematic study of how each component affects performance and compare reinitializing units against weights. These studies reveal limitations in existing selective reinitialization methods and provide solutions to address them. Collectively, this work establishes selective reinitialization as a robust and general approach for maintaining plasticity across deep learning systems.

This dissertation systematically and incrementally builds knowledge about plasticity loss and selective reinitialization. The next chapter introduces background knowledge on deep learning systems as well as the continual supervised learning problem, which will be central throughout. Chapter 3 introduces the concept of loss of plasticity, establishing it as a widespread phenomenon in learning systems based on fully-connected networks. Chapter 4 and Chapter 5 introduce the idea of selective reinitialization at the unit and weight level, respectively. Together, these chapters demonstrate that selective reinitialization is a robust approach for maintaining plasticity in fully-connected networks. Chapters 6 and 7 extend the ideas in the previous three chapters to convolutional neural networks and modern architectures, respectively. These two chapters show that plasticity loss persists across diverse architectures and that selective reinitialization remains effective. The final chapter summarizes the dissertation's contributions and results and discusses avenues for future work.

Chapter 2

Background

This thesis contributes to the scientific field dedicated to designing and training artificial neural networks, also referred to as *deep learning*. This chapter reviews the foundational deep learning concepts required for understanding the contributions of this thesis. The first section in this chapter presents the mathematical notation used in this and the following chapters. The next section defines the continual supervised learning problem, the main learning problem studied in this thesis. The third section presents ideas related to neural networks and the stochastic optimization algorithms used for training them. Lastly, the last section in this chapter discusses the different types of layers and architectures used in deep learning systems. The ideas about neural networks and stochastic approximation will be relevant in future chapters, as they form the core of the different learning systems studied in this dissertation.

2.1 Notation

In this dissertation, the set of natural number is denoted $\mathbb{N}$ or $\mathbb{N}_0$ when including zero; the set of real numbers is denoted $\mathbb{R}$; and the set containing elements $\{1, 2, 3, \ldots, n\}$, for some $n \in \mathbb{N}$, is denoted $[n]$. Bold lowercase letters are used to denote vectors, whereas matrices or tensors are denoted with bold uppercase letters. For example, $\mathbf{x} \in \mathbb{R}^n$ corresponds to a vector with n real elements, whereas $\mathbf{W} \in \mathbb{R}^{m \times d}$ corresponds to a real-valued matrix with m rows and d columns. Individual elements of vectors or matrices are denoted using brackets containing the index or indices of the elements using a comma-separated list. For example, $\mathbf{W}[i, j]$ is the element in the i-th column and j-th row of matrix $\mathbf{W}$. Multiple sequential elements in a matrix or vector are denoted using brackets containing two indices separated by a colon. For example, if $\mathbf{W} \in \mathbb{R}^{m \times d}$, then $\mathbf{W}[a{:}b,\ q{:}r]$ is a submatrix with dimensions $(b - a) \times (r - q)$, assuming $m \geq b > a$, $d \geq r > q$. A single colon with no indices references an entire dimension in the matrix.

Scalar random variables are denoted using an uppercase X or Y, whereas n-dimensional random variables are denoted with a bold uppercase $\mathbf{X}$ or $\mathbf{Y}$. The use of p is reserved to denote the distribution of a random variable. For example, a random variable X with distribution p is denoted as $X \sim p$. p can also correspond to a joint probability distribution. Some of the most common parameterized probability distributions have their own special notation. The normal probability distribution with mean, μ, and standard deviations, σ, is denoted $\mathcal{N}(\mu, \sigma)$, whereas the uniform distribution with lower bound, a, and upper bound, b is denoted $\mathcal{U}(a, b)$. The expected value and variance of a random variable, X, with probability distribution p are denoted $\mathbb{E}_p[X]$ and $\mathbb{V}_p[X]$, respectively.

Other uses of subscripts are reserved to denote elements of sequences or parameters of functions. For example, for a sequence of random variables $\{X_i\}_{i=1}^n$, X_i is the i-th element in the sequence. The subscript t is reserved to denote the current time step in iterative algorithms. Calligraphic lowercase letters are used to denote functions, e.g., f and g. For parameterized functions other than the Normal and Uniform distributions mentioned above, the subscript of the function indicates the parameters of the function. For example, the function f parameterized by θ is denoted f_θ.

2.2 Continual Supervised Learning of Tasks

The central learning problem studied in this dissertation is continual supervised learning. In continual supervised learning, a learning system is required to generate predictions about a desired target based on a stream of non-stationary data. Formally, the learning system is tasked with generating predictions, $\hat{\mathbf{Y}} \in \mathbb{R}^c$, based on observations, $\mathbf{X} \in \mathbb{R}^n$, to match some desired target, $\mathbf{Y} \in \mathbb{R}^c$, where $c, d \in \mathbb{N}$. Pairs of observations and targets are presented one at a time, forming a sequence $\{(\mathbf{X}_t, \mathbf{Y}_t)\}_{t \geq 0}$. Observation-target pairs are jointly distributed according to a probability distribution p_k, $k \in \mathbb{N}$, which changes every certain number of time steps. The change in the data-generating process is the source of non-stationarity. It is convenient to think of a subsequence of consecutive observation-target pairs generated by the same distribution as a *task*. The number of observation-target pairs presented during a task is denoted τ_k.

The mismatch between predictions and targets is measured according to a loss function, ℓ, which takes a prediction and a target, $(\hat{\mathbf{Y}}, \mathbf{Y})$, and returns a real value in the interval $[0, \infty)$. The loss function indicates how inaccurate the predictions are, providing a signal that can be used for improving future predictions. All the experiments in this thesis use the cross-entropy loss, which is defined as:

$$\ell(\hat{\mathbf{Y}}, \mathbf{Y}) \doteq -\sum_{i=1}^{c} \mathbf{Y}[i] \cdot \log\left(\hat{\mathbf{Y}}[i]\right), \tag{2.1}$$

where it is assumed that entries in $\mathbf{Y}$ and $\hat{\mathbf{Y}}$ are vectors of probabilities whose entries add up to

one.

A learning system consists of two components: an approximation function, f, that generates predictions and an optimization algorithm, $\mathcal{O}$, that updates the function. This work considers only parameterized approximation functions that use observations, $\mathbf{X}_t$, and a parameter vector, $\boldsymbol{\theta}_t \in \mathbb{R}^d$, to generate predictions, $\mathbf{Y}_t$. To update the parameterized function, the approximation algorithm updates the parameter vector using the current observation-target pair and the loss function, resulting in a new parameter vector $\boldsymbol{\theta}_{t+1}$. The process of updating the parameter vector generates a sequence of iterates, $\{\boldsymbol{\theta}_t\}_{t\geq 0}$, starting from a random parameter vector $\boldsymbol{\theta}_0$.

The goal of a learning system is to improve its predictions continually. Ideally, we would like learning systems that minimize

$$J(\boldsymbol{\theta}_t) \doteq \mathbb{E}_{p_k}\left[\ell\left(f_{\boldsymbol{\theta}_t}(\mathbf{X}_t), \mathbf{Y}_t\right)\right], \tag{2.2}$$

where $\mathbb{E}_{p_k}$ is the expectation under the current sampling distribution and $f_{\boldsymbol{\theta}_t}$ is the approximation function parameterized by $\boldsymbol{\theta}_t$. However, access to p_k is rarely available in practice. Instead, learning systems are designed to minimize the *empirical loss* computed from samples. Let $\{(\mathbf{x}_{t,i}, \mathbf{y}_{t,i})\}_{i=1}^m$ be a mini-batch of m observation-target pairs sampled at time t from p_k. The empirical loss is defined as

$$\hat{J}(\boldsymbol{\theta}_t) \doteq \frac{1}{m}\sum_{i=1}^{m} \ell\left(f_{\boldsymbol{\theta}_t}(\mathbf{x}_{t,i}), \mathbf{y}_{t,i}\right). \tag{2.3}$$

The measure of performance of a learning system in a given task is the *average expected loss*. Let $T_j = \sum_{i=1}^{j-1} \tau_i$ be the total number of time steps transpired before task j, with $T_1 \doteq 0$. The average expected loss for the k-th task is

$$\mathcal{L}_k(\boldsymbol{\theta}_{T_k}) \doteq \frac{1}{\tau_k}\sum_{j=0}^{\tau_k-1} \mathbb{E}_{p_k}\left[\ell\left(f_{\boldsymbol{\theta}_{T_k+j}}(\mathbf{X}_{T_k+j}), \mathbf{Y}_{T_k+j}\right)\right]. \tag{2.4}$$

Once again, we face the problem of not having access to the distribution function p_k. In practice, we instead measure the *average empirical loss*:

$$\hat{\mathcal{L}}_k(\boldsymbol{\theta}_{T_k}) \doteq \frac{1}{\tau_k}\sum_{j=0}^{\tau_k-1}\frac{1}{m}\sum_{i=1}^{m} \ell\left(f_{\boldsymbol{\theta}_{T_k+j}}(\mathbf{x}_{T_k+j,i}), \mathbf{y}_{T_k+j,i}\right). \tag{2.5}$$

This specific formulation of continual supervised learning favours algorithms that can learn rapidly from each task. As a consequence, algorithms that can exploit shared structure across tasks are expected to perform well. On the other hand, algorithms designed to prevent catastrophic forgetting would only have an advantage if tasks repeat. Finally, algorithms that lose their ability to learn over time would perform poorly as the number of tasks increases, especially if tasks are

equally difficult to learn and do not repeat. Hence, this problem formulation is particularly well posed for identifying learning systems that lose plasticity over time.

2.3 Function Approximation With Neural Networks and Stochastic Gradient Descent

One of the key questions in the continual supervised learning problem is how to find an approximation function, f, that minimizes the empirical loss in (2.3). When using parameterized functions, this search is equivalent to finding a parameter vector, $\boldsymbol{\theta}$, that minimizes the empirical loss.

The approach in this work is to use approximation functions based on artificial neural networks. An artificial neural network is a composition of several nonlinear functions, each parameterized by its own set of parameters. The functional form of the network, colloquially referred to as the *network architecture*, specifies a computational graph that determines how observations are processed to generate predictions. The design of network architectures is discussed in the next section. Although the parameters of the network may be distributed among several different functions, it is convenient to still denote them as a single parameter vector, $\boldsymbol{\theta}$.

The network's parameters can be adjusted to minimize the loss between predictions and targets. Finding the appropriate parameter values is the role of the optimization algorithm, $\mathcal{O}$, and is a process often referred to as *learning*. A common optimization algorithm for learning network parameters is mini-batch stochastic gradient descent. Mini-batch stochastic gradient descent, or SGD for short, is a local search method that iteratively updates the parameters of the network using the update rule:

$$\boldsymbol{\theta}_{t+1} \doteq \boldsymbol{\theta}_t - \alpha_t \cdot \frac{1}{m} \sum_{i=1}^{m} \nabla_{\boldsymbol{\theta}_t} \ell(f_{\boldsymbol{\theta}_t}(\mathbf{x}_{t,i}), \mathbf{y}_{t,i}), \tag{2.6}$$

where $\nabla_{\boldsymbol{\theta}_t} \ell(f_{\boldsymbol{\theta}_t}(\mathbf{x}_{t,i}), \mathbf{y}_{t,i})$ is the derivative of the loss with respect to the current parameters and $\alpha_t > 0$ is a step-size parameter. If ℓ is convex in $\boldsymbol{\theta}$, the SGD update rule is guaranteed to converge to the global minimum if $\sum_{t=1}^{\infty} \alpha_t = \infty$, $\sum_{t=1}^{\infty} \alpha_t^2 < \infty$, and the observation-target pairs are sampled from a stationary probability distribution. However, due to the nonlinearity of neural networks, ℓ is typically not convex in $\boldsymbol{\theta}$, so these algorithms can only be expected to converge to a local minimum.

There are several modifications to the SGD update. A common modification is parameter regularization, which constrains the parameter norm from growing too large. Constraining the norm of the parameters is a common approach in deep learning for preventing overfitting (see Chapter 7 of Goodfellow et al., 2016). The most common regularization approach is L2 regularization, also known as weight decay, which adds a penalty term to the loss proportional to the L2 norm of the

parameter vector. The corresponding update function is:

$$\boldsymbol{\theta}_{t+1} \doteq (1 - \alpha_t \lambda) \cdot \boldsymbol{\theta}_t - \alpha_t \cdot \frac{1}{m} \sum_{i=1}^{m} \nabla_{\boldsymbol{\theta}_t} \ell(f_{\boldsymbol{\theta}_t}(\mathbf{x}_{t,i}), \mathbf{y}_{t,i}), \tag{2.7}$$

where $\lambda > 0$ is a regularization factor.

Another common modification to the SGD update is to use a moving average of the gradient rather than the gradient computed on the current mini-batch of data. The corresponding optimization algorithm is also known as SGD with momentum, and uses the following two equations to update the parameters of the network:

$$\begin{aligned} \boldsymbol{m}_{t+1} &= (1 - \beta) \cdot \boldsymbol{m}_t + \beta \cdot \left(\frac{1}{m} \sum_{i=1}^{m} \nabla_{\boldsymbol{\theta}_t} \ell(f_{\boldsymbol{\theta}_t}(\mathbf{x}_{t,i}), \mathbf{y}_{t,i}) + \lambda \cdot \boldsymbol{\theta}_t \right) \\ \boldsymbol{\theta}_{t+1} &= \boldsymbol{\theta}_t - \alpha_t \cdot \boldsymbol{m}_{t+1} \end{aligned} \tag{2.8}$$

where $\beta \in [0, 1]$ is a moving average factor and $\lambda \cdot \theta_t$ is only added when using L2 regularization. The Adam optimizer takes this idea one step further by also scaling the update by a moving average of the square of the gradients (Kingma and Ba, 2015), resulting in the update:

$$\begin{aligned} \boldsymbol{g}_{t+1} &= \frac{1}{m} \sum_{i=1}^{m} \nabla_{\boldsymbol{\theta}_t} \ell(f_{\boldsymbol{\theta}_t}(\mathbf{x}_{t,i}), \mathbf{y}_{t,i}) + \lambda \cdot \boldsymbol{\theta}_t \\ \boldsymbol{m}_{t+1} &= (1 - \beta_1) \cdot \boldsymbol{m}_t + \beta_1 \cdot \boldsymbol{g}_{t+1} \\ \boldsymbol{v}_{t+1} &= (1 - \beta_2) \cdot \boldsymbol{v}_t + \beta_2 \cdot \boldsymbol{g}_{t+1}^2 \\ \hat{\boldsymbol{m}} &= \frac{\boldsymbol{m}_{t+1}}{(1 - \beta_1^{t+1})}, \quad \hat{\boldsymbol{v}} = \frac{\boldsymbol{v}_{t+1}}{(1 - \beta_2^{t+1})} \\ \boldsymbol{\theta}_{t+1} &= \boldsymbol{\theta}_t - \alpha_t \cdot \frac{\hat{\boldsymbol{m}}}{\sqrt{\hat{\boldsymbol{v}}} + \epsilon}, \end{aligned} \tag{2.9}$$

where $\beta_1, \beta_2 \in [0, 1]$ are moving average factors, $\boldsymbol{g}_{t+1}^2$ is the square function applied elementwise to $\boldsymbol{g}_{t+1}$, $\epsilon > 0$ is a small number to prevent division by zero, and $\lambda \cdot \boldsymbol{\theta}_t$ is a term added only when using L2 regularization. Lastly, Loshchilov and Hutter (2019) suggested that including a regularization term in the moving averages may lead to optimization instabilities. The authors proposed the SGDW optimizer with update:

$$\begin{aligned} \boldsymbol{m}_{t+1} &= (1 - \beta) \cdot \boldsymbol{m}_t + \beta \cdot \left(\frac{1}{m} \sum_{i=1}^{m} \nabla_{\boldsymbol{\theta}_t} \ell(f_{\boldsymbol{\theta}_t}(\mathbf{x}_{t,i}), \mathbf{y}_{t,i}) \right) \\ \boldsymbol{\theta}_{t+1} &= (1 - \alpha_t \lambda) \cdot \boldsymbol{\theta}_t - \alpha_t \cdot \boldsymbol{m}_{t+1}, \end{aligned} \tag{2.10}$$

and the AdamW optimizer with update:

$$\begin{aligned} \boldsymbol{g}_{t+1} &= \frac{1}{m}\sum_{i=1}^{m} \nabla_{\boldsymbol{\theta}_t} \ell(f_{\boldsymbol{\theta}_t}(\mathbf{x}_{t,i}), \mathbf{y}_{t,i}) \\ \boldsymbol{\theta}_{t+1} &= (1 - \lambda\alpha_t) \cdot \boldsymbol{\theta}_t - \alpha_t \cdot \frac{\hat{\boldsymbol{m}}}{\sqrt{\hat{\boldsymbol{v}}} + \epsilon}, \end{aligned} \tag{2.11}$$

where $\hat{\boldsymbol{m}}$ and $\hat{\boldsymbol{v}}$ are defined as in the Adam optimizer update in (2.9).

2.4 Neural Network Architectures

A neural network architecture is a composition of nonlinear functions, each parameterized by its own set of parameters. Each function is often referred to as a *layer*. In other words, a neural network architecture is composed of several layers. Layers contain several *units* of computation that differ depending on the type of layer.

The simplest type of layer is a *linear* layer which applies an affine transformation to its input. Formally, for an input $\mathbf{x} \in \mathbb{R}^n$, a matrix of parameters $\mathbf{W} \in \mathbb{R}^{m \times n}$ and a vector of parameters $\mathbf{b} \in \mathbb{R}^m$, a linear layer applies the transformation:

$$\text{Linear}(\mathbf{x}) = \mathbf{W}\mathbf{x} + \mathbf{b}, \tag{2.12}$$

The the elements of the parameter matrix, $\mathbf{W}$, are also known as the weights of the layer, whereas the elements of the vector, $\mathbf{b}$, are known as the biases. The transformation in the equation above is affine. To make the transformation nonlinear, layers often include a nonlinear function, $g : \mathbb{R} \to \mathbb{R}$, that is applied elementwise to the output of the affine transformation:

$$\text{Fully-Connected}(\mathbf{x}) = g(\mathbf{W}\mathbf{x} + \mathbf{b}). \tag{2.13}$$

The nonlinear function, g, is often referred to as the *activation function*. A multitude of nonlinear activation functions are used in practice. This dissertation considers a selection of the most commonly used activation functions, which are illustrated in Figure 2.1. The outputs of the affine transformation before the nonlinear function are often referred to as the *pre-activations*, whereas the outputs of the nonlinear function are known as the *activations*, which are denoted $\boldsymbol{h}$. I refer to linear layers that include a nonlinear activation function as *fully-connected* layers.

Another type of layer used in this dissertation is the convolutional layer. Convolutional layers gained popularity due to their success at image recognition problems (Lecun et al., 1998). The inputs to convolutional layers are multi-dimensional tensors, such as images. The layer consists of several filters that slide over the image, applying the same nonlinear function to different patches

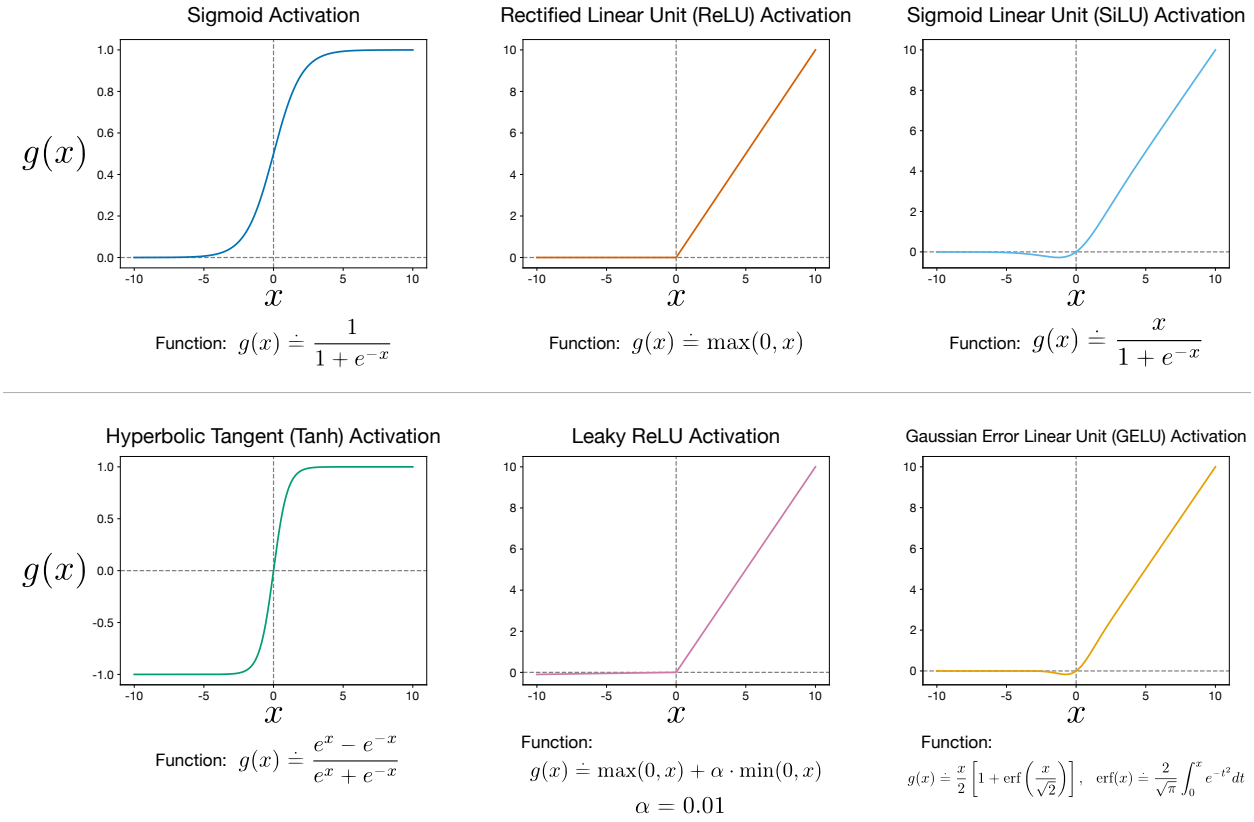

Figure 2.1: Different types of nonlinear activation functions. This represents only a small sample of the activation functions used in deep learning systems.

and producing multi-dimensional outputs, called channels, denoted $\boldsymbol{H}$. A single filter in the layer consists of a tensor of weights, for example $\mathbf{W} \in \mathbb{R}^{d \times d}$, and a scalar bias, e.g., $b \in \mathbb{R}$. Given an input $\boldsymbol{A} \in \mathbb{R}^{n \times n}$ with $n > d$, the filter is applied to submatrices of $\boldsymbol{A}$, $B \in \mathbb{R}^{d \times d}$, using the following equation:

$$f(\boldsymbol{B}) = g\left(b + \sum_{i=1}^{d}\sum_{j=1}^{d} \mathbf{W}[i,j] \cdot \boldsymbol{B}[i,j]\right). \tag{2.14}$$

The number of submatrices processed in such a manner depends on a hyperparameter called *stride*, which controls how fast the filter slides across the image. Often, to preserve the input's dimensions, the input is padded with zeros around its edges. The amount of padding is controlled by a hyperparameter called *padding*. The final dimensions of a channel depend on the stride and padding values. Each filter in the convolutional layer has its own set of weights and bias, and produces one output channel. The dimensions of the output of a convolutional layer, $\boldsymbol{H}$, are $c \times d_1 \times d_1$, where c is the number of filters in the layer and d_1 corresponds to the dimensions of the channels.

Convolutional layers are often used in conjunction with pooling layers. Pooling layers apply

a pooling operation, such as the average or the maximum, over two-dimensional patches of the channels of convolutional layers. The pooling layers slide over the channels like convolutional filters. Depending on the stride and padding, pooling layers may reduce the dimensions of channels.

Another type of layer that is popular in natural language processing applications is the attention layer. This type of layer was initially proposed by Bahdanau et al. (2014) and was popularized by the transformer architecture proposed by Vaswani et al. (2017), which is now central to large language model applications. A central part of the layer is the use of the SoftMax function, which for vector $\mathbf{x} \in \mathbb{R}^n$, performs the operation

$$\text{SoftMax}(\mathbf{x}) \doteq \left[\frac{e^{\mathbf{x}[1]}}{\sum_{j=1}^{n} e^{\mathbf{x}[j]}}, \cdots, \frac{e^{\mathbf{x}[n]}}{\sum_{j=1}^{n} e^{\mathbf{x}[j]}} \right]^\top, \tag{2.15}$$

essentially transforming a vector of real values into a vector of probabilities. Attention layers take as input a matrix $\mathbf{X} \in \mathbb{R}^{n \times d}$ and creates query, keys, and value matrices using matrices of learnable parameters:

$$\begin{aligned} \boldsymbol{Q} &\doteq \mathbf{X}\mathbf{W}_Q, \quad \mathbf{W}_Q \in \mathbb{R}^{d \times d_Q} \\ \boldsymbol{K} &\doteq \mathbf{X}\mathbf{W}_K, \quad \mathbf{W}_K \in \mathbb{R}^{d \times d_Q} \\ \boldsymbol{V} &\doteq \mathbf{X}\mathbf{W}_V, \quad \mathbf{W}_V \in \mathbb{R}^{d \times d_V}. \end{aligned}$$

These matrices are used to perform the operation

$$\text{Attention}(\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}) \doteq \text{SoftMax}\left(\frac{\boldsymbol{Q}\boldsymbol{K}^\top}{\sqrt{d}} \right) \boldsymbol{V}, \tag{2.16}$$

where the SoftMax function is applied over the rows of the matrix. The function in (2.16) is also known as self-attention since the query, keys, and values are all computed from the same input. To increase the diversity of representations learned by the attention layer, it is common to concatenate several attention heads. The resulting operation is known as multi-head attention, which is given by

$$\text{Multi-Head Attention}(\mathbf{X}) \doteq \text{Concatenate}(\text{head}_1, \cdots, \text{head}_h)\mathbf{W}_O, \quad \mathbf{W}_O \in \mathbb{R}^{d_V \cdot h \times d} \tag{2.17}$$

$$\text{head}_i \doteq \text{Attention}\left(\mathbf{X}\mathbf{W}_{Q,i}, \mathbf{X}\mathbf{W}_{K,i}, \mathbf{X}\mathbf{W}_{V,i}\right), \tag{2.18}$$

where h is the number of attention heads, $\mathbf{W}_O$, $\mathbf{W}_{Q,i}$, $\mathbf{W}_{K,i}$, and $\mathbf{W}_{V,i}$ are matrices of learnable parameters. Note that the multi-head attention layer preserves the input dimensions.

A common technique used along attention layers is dropout (Hinton et al., 2012), which randomly sets inputs to zero with a probability of $q \in [0, 1)$. For a given input, $\mathbf{x} \in \mathbb{R}^n$, dropout creates a mask, $\mathbf{m} \in \mathbb{R}^n$, with entries set to zero with probability q and one with probability $1 - q$.

During training, dropout performs the operation:

$$\text{Dropout}(\mathbf{x}) \doteq \mathbf{m} \ \odot \mathbf{x} \cdot (1-q)^{-1},$$

where $\odot$ is the element-wise multiplication of two vectors. During testing, dropout performs the identity operation.

A standard technique in state-of-the-art deep learning systems is using normalization on the output of hidden layers. There are multiple approaches for computing the statistics used in the normalization operation. Batch norm, proposed by Ioffe and Szegedy (2015), computes the statistics based on a mini-batch of data. Given a mini-batch of observations $\{\mathbf{x}_i\}_{i=1}^m$, batch norm computes the sample average, $\overline{\mathbf{x}} \doteq \frac{1}{m}\sum_{i=1}^m \mathbf{x}_i$, and sample standard deviation, $\mathbf{S}^2 \doteq \frac{1}{m}\sum_{i=1}^m (\mathbf{x}_i - \overline{\mathbf{x}})^2$, where the square function is applied element-wise. Given these two statistics, the inputs are normalized by

$$\text{Batch Norm}(\mathbf{x}) \doteq (\mathbf{x} - \overline{\mathbf{x}}) \odot (\mathbf{S}^2 + \epsilon)^{-\frac{1}{2}} \odot \boldsymbol{\gamma} + \boldsymbol{\beta}, \tag{2.19}$$

where $\epsilon > 0$ is a small number to prevent division by zero, the square root is applied element-wise, and $\boldsymbol{\gamma}$ and $\boldsymbol{\beta}$ are learnable scaling and translation parameters, respectively. An alternative to using a mini-batch of data is to compute the statistics over the elements of the input itself. The corresponding normalization operation is known as layer norm, which was proposed by Ba et al. (2016). Let $\overline{\mathrm{x}} \doteq \frac{1}{n}\sum_{i=1}^n \mathbf{x}[i]$ and $\mathrm{s}^2 = \frac{1}{n}\sum_{i=1}^n (\mathbf{x}[i] - \overline{\mathrm{x}})^2$. Layer normalization performs the operation

$$\text{Layer Norm}(\mathbf{x}) \doteq \frac{\mathbf{x} - \overline{\mathrm{x}}}{\sqrt{\mathrm{s}^2 + \epsilon}} \odot \boldsymbol{\gamma} + \boldsymbol{\beta}. \tag{2.20}$$

All the layers discussed so far can be combined sequentially to form a neural network architecture. When talking about multiple layers in a network, it will be helpful to think of $\boldsymbol{\theta}$ as an ordered set containing the weights of each layer in sequence. For example, for a network with a fully-connected layer and a linear output layer without bias terms, $\boldsymbol{\theta}$ would be the set $\{\mathbf{W}_0, \mathbf{W}_1\}$. For a given input $\mathbf{x}$ and activation function g, the network in the example performs the operation:

$$f_{\boldsymbol{\theta}}(x) = \mathbf{W}_1 \ g(\mathbf{W}_0 \mathbf{x}).$$

An important concept that will be relevant in future chapters is the idea of a unit of computation in a layer. Units of computation have corresponding input weights that connect the unit to previous layers, and output weights that connect the unit to the following layers. In the example above, $\mathbf{W}_0$ are the input connections to the fully-connected layer, whereas $\mathbf{W}_1$ are the output connections. This concept will be important in Chapter 4 when talking about unit reinitialization.

The definition of a unit requires identifying its input and output connections, as well as the value of the unit when processing observations. Such a task becomes increasingly intricate as the network architecture includes several different types of layers. We will revisit the definition of a unit several times as different architectures are introduced throughout this dissertation.

Chapter 3

Loss of Plasticity in Fully-Connected Networks

This chapter begins the systematic demonstrations of plasticity loss in this dissertation, focusing on fully-connected networks. These demonstrations include a broad range of algorithmic choices typical in deep learning systems as well as prior techniques for stabilizing learning. To facilitate this extensive study, experiments are conducted on a continual classification problem based on the MNIST dataset.

The results establish that diverse deep learning configurations are susceptible to plasticity loss. However, the phenomenon is not inevitable: techniques such as shrink-and-perturb (Ash and Adams, 2020) and L2 regularization successfully prevent it in fully-connected networks, though their effectiveness in other architectures remains an open question explored in later chapters.

This work extends experiments from collaborative research published in Nature (Dohare et al., 2024) and the Conference on Lifelong Learning Agents (Hernandez-Garcia et al., 2025), providing deeper characterization of plasticity loss in fully-connected architectures.

3.1 The Permuted MNIST Problem and How to Evaluate Loss of Plasticity

The MNIST dataset consists of 28×28 grayscale images of digits from 0 to 9, along with their corresponding labels (LeCun et al., 2010). Each pixel can take integer values between 0 and 255, but in this work, they are normalized to $[0, 1]$ by dividing by 255. The dataset contains 60,000 images for training and 10,000 for testing, with nearly equal numbers of images per digit. By today's standards, it is a small dataset that current deep learning systems can easily learn to classify

with high accuracy. Nevertheless, the dataset's size makes it ideal for thorough and systematic experimentation. Moreover, with careful experimental design, the dataset can still provide valuable insights into the inner workings of deep learning systems, making it one of the most widely used datasets in deep learning research.

The Permuted MNIST problem is a continual supervised learning problem consisting of a sequence of classification tasks, each based on a random permutation of the pixels of the images in MNIST. To generate a task, a random permutation of the pixels in an image is sampled and applied to every image in the dataset. Applying the same permutation to the pixels of each image preserves the mapping from images to labels. Consequently, it should still be possible to learn to correctly classify the permuted images. However, any learning system that had learned to classify MNIST images before permuting the pixels would see its performance severely decreased.

To maintain good performance on the Permuted MNIST problem, learning systems must continually relearn the mapping from images to labels after each permutation. Learning systems are trained for a single pass through the dataset per task for several tasks in a process described in Figure 3.1a. This experimental design was first introduced by Goodfellow et al. (2014) to investigate catastrophic forgetting. In this dissertation, the Permuted MNIST problem is repurposed to study the ability to learn of systems trained on non-stationary data.

The system's accuracy is the main indicator of how well it is learning to classify images. Accuracy is computed on each mini-batch of data before updating the network's parameters; such a measurement is called the *online accuracy*. The performance metric for the learning system across the entire task is the online accuracy averaged over all mini-batches, or simply the *average online accuracy*. The question of interest is: how does the average online accuracy change as the system continues to learn from new tasks?

To answer this question, let us imagine three hypothetical scenarios we might encounter as the system learns for multiple tasks. Let us assume that all tasks are equally difficult to classify. If the system does not lose plasticity, then we should expect its performance to remain roughly the same across different tasks, as illustrated by the blue line in Figure 3.1b. Alternatively, if a system does not lose plasticity, it may exploit the shared structure between tasks, increasing performance across tasks—also known as *positive transfer*—as illustrated by the yellow line in Figure 3.1b. However, if the system slowly loses plasticity, we would expect performance to decrease gradually as new tasks are introduced, as illustrated by the black line in Figure 3.1b.

The three scenarios in Figure 3.1b are all hypothetical. To investigate what actually happens, let us observe the performance of a learning system based on a fully-connected network. Note that, for such a system, each task should be equally difficult, since fully-connected layers do not exploit spatial information between pixels. The network includes three fully-connected layers with 100 ReLU units each. The learning system minimizes the cross-entropy loss between predictions

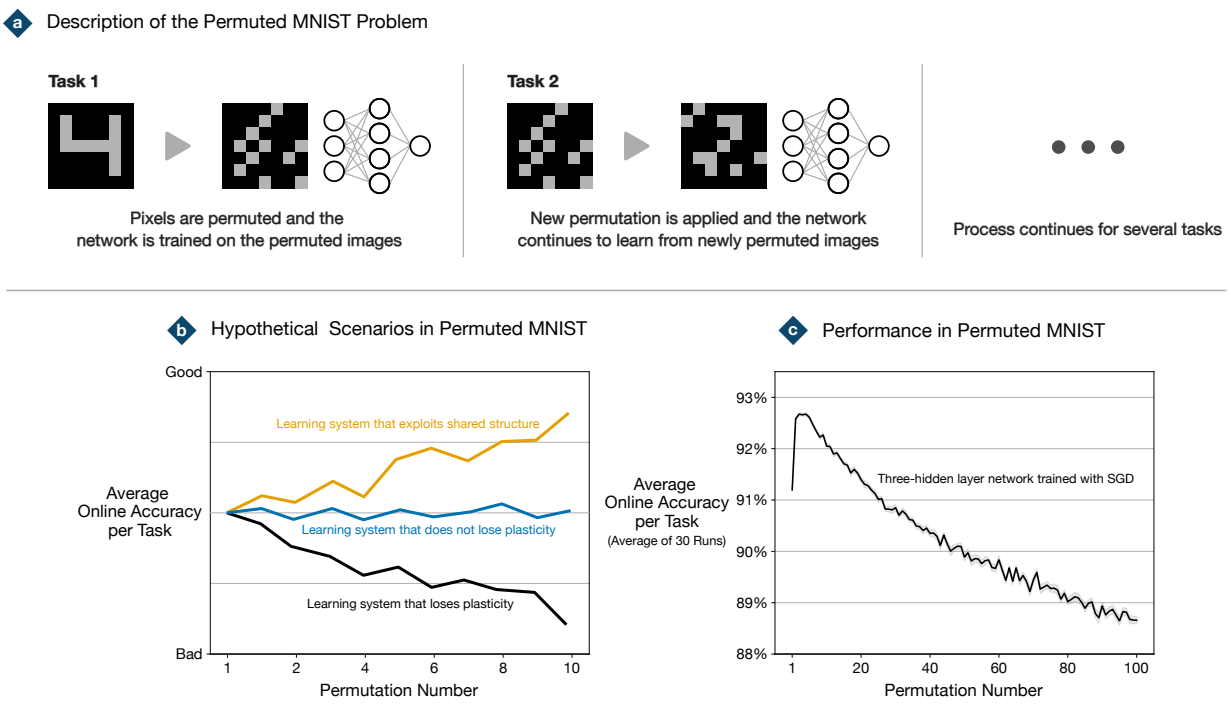

Figure 3.1: **a.** Description of the Permuted MNIST problem. **b.** Hypothetical scenarios of the performance of learning systems depending on whether they lose plasticity. **c.** Performance of a three-hidden-layer network. The line represents the average of 30 independent runs, and the shaded regions indicate the range of one standard error above and below the average.

and labels using stochastic gradient descent (SGD) with a step-size of 0.05 using a mini-batch size of 30. The weight matrices of the network are initialized using Kaiming Normal initialization (He et al., 2015) and the bias vectors are initialized to zero.

Figure 3.1c shows the results of the experiment, averaged across 30 independent runs, with shaded regions indicating the standard error. The learning curve does not match any of the hypothetical scenarios but shows the characteristics of all three. Initially, the performance increased, indicating some positive transfer across the first few tasks. Then, the performance plateaued for a few tasks, followed by a persistent decline, indicating a loss of the ability to learn. The performance continued to decrease steadily until the end of the experiment and may have continued this trend had the experiment been run for a longer sequence of tasks.

The following section demonstrates a similar pattern in the performance of a wide variety of learning systems based on fully connected networks.

3.2 Demonstrations of Plasticity Loss With Fully-Connected Networks

The results in the previous section provide evidence that plasticity is not guaranteed in deep learning systems when networks are trained continually. However, those results do not provide a sense of how widespread the problem is. This section demonstrates that several configurations of deep learning systems are susceptible to plasticity loss. The goal is to provide evidence that loss of plasticity is a widespread phenomenon encountered in various deep learning systems.

One possible explanation for the poor performance shown in Figure 3.1c is a poor choice of hyperparameters. Perhaps we could find a value for the step-size parameter for SGD that does not result in such a drastic decrease in performance. To answer this question, the same learning system is trained using different values of the step-size parameter. The results in Figure 3.2a show that the performance of every system eventually declined. For the smallest step-size value, the decrease in performance was slow, indicating that plasticity loss was significantly reduced. However, this choice of step-size also resulted in low accuracy, making it a less desirable option than the alternatives. High values of the step-size parameter yielded good performance, albeit at the cost of greater plasticity loss, whereas low values slowed plasticity loss but resulted in lower performance.

Hyperparameter tuning will be a recurrent concern in all the experiments presented in this dissertation. Generally, the procedure for tuning hyperparameters consists of three steps. First, a wide range of hyperparameter values is tested for a single run to find a sensible range where the algorithm learns reasonably well. In this context, learning reasonably well means that the optimization process does not diverge and the performance is better than random. After finding a suitable range of values, the second step is to perform a grid search using multiple independent runs (between five and ten runs). The third and final step is to identify the hyperparameter combination that yields the highest average performance, also referred to as the *best hyperparameter combination*. After identifying the best hyperparameter combination, the algorithm is run for several additional independent runs, typically reaching a total of 15 to 30 runs. The number of runs used for the grid search and the last step are chosen based on the problem's inherent noise and the cost of running each experiment. In the main text, the specific hyperparameter are omitted except where knowing them is relevant for analysis. Appendix A provides complete details about the specific hyperparameter values tested.

The step-size is not the only algorithmic choice that can have a drastic impact on performance. It is conventional wisdom in deep learning that scaling up the number of parameters in learning systems can lead to significant performance improvements (Kaplan et al., 2020). Perhaps we could see a similar effect in the Permuted MNIST problem, resulting in the mitigation or complete removal of plasticity loss. To test this possibility, the following evaluation presents the performance

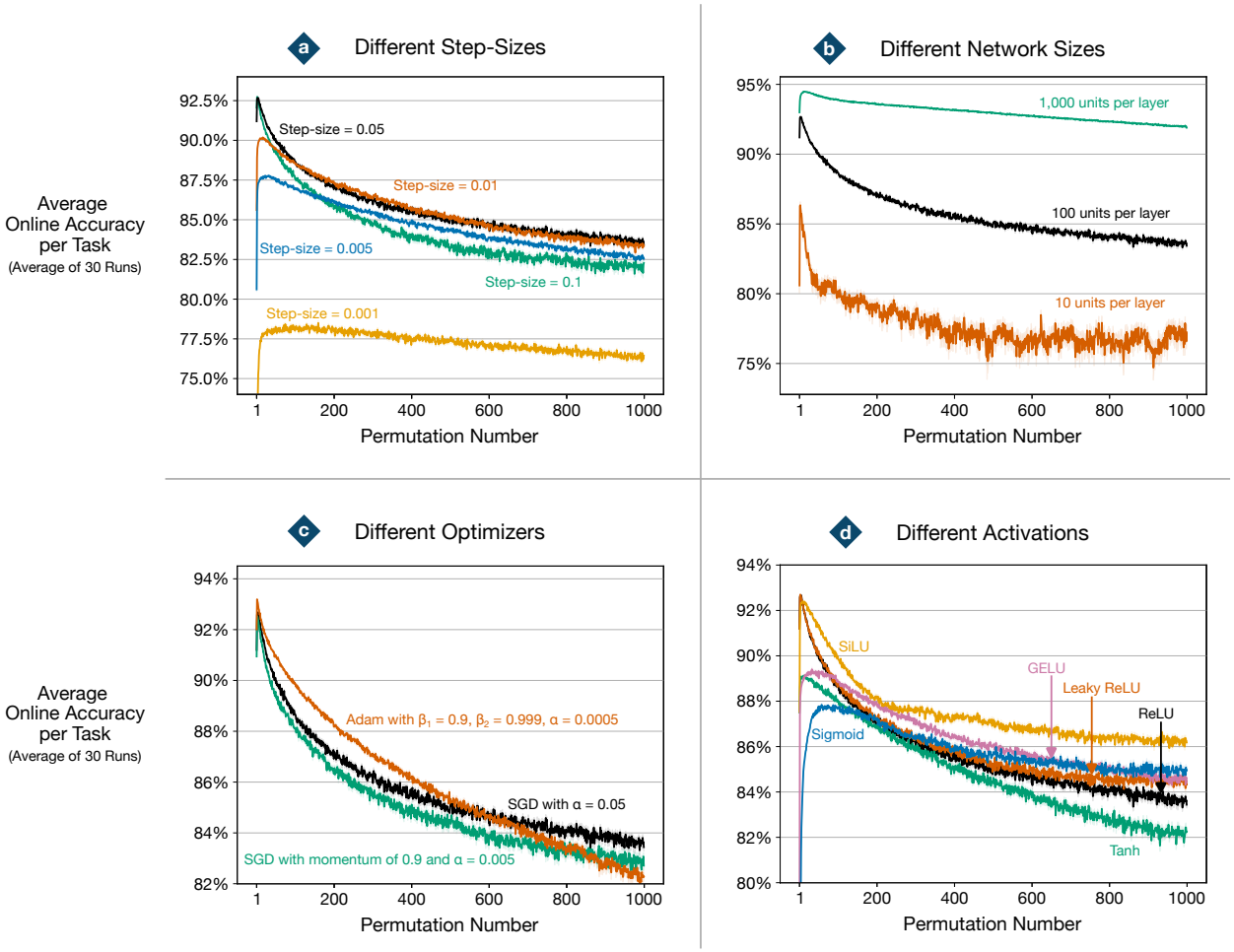

Figure 3.2: **a.** Plasticity loss with different step-sizes. **b.** Plasticity loss with varying sizes of networks. **c.** Plasticity loss with different optimizers. **d.** Plasticity loss with different activation functions. All results are the average of 30 runs, with shaded regions representing one standard error away from the mean. Table A.1 lists the hyperparameters used in this figure.

of systems with three fully-connected layers, each with ReLU activations and 10, 100, or 1,000 units per layer. The step-size of SGD was tuned to maximize the average online accuracy over all $1,000$ tasks in the experiment.

Increasing the size of the network resulted in higher average online accuracy and a slower drop in performance as the number of tasks increased (Figure 3.2b). Nevertheless, all the networks still showed some amount of plasticity loss. It is likely that further increasing the number of units would completely prevent plasticity loss within the $1,000$ tasks. However, such a solution would entail higher computational costs and may not resolve the problem for a larger number of tasks.

The next question is whether the optimization algorithm is responsible for the performance

drop observed in the Permuted MNIST problem. To answer this question, the next evaluation includes learning systems using SGD, SGD with momentum, and the Adam optimizer. The network architecture comprises three fully-connected layers, each with 100 ReLU units. For SGD with momentum, the momentum term and the step-size were tuned. For Adam, the moving average factors, β_1 and β_2, were tuned simultaneously with the step-size.

Regardless of the optimization algorithm, the performance showed a similar pattern as before (Figure 3.2c). Performance initially rose during the first few tasks, then decreased for the rest of the experiment. The steady decrease in performance suggests that the three different learning systems experienced a loss of plasticity.

An architectural choice that often has a drastic impact on performance is the activation function. The results so far have shown the performance of networks with ReLU units. The following evaluation tests if plasticity loss is present when using other activation functions. The following experiment uses Sigmoid, Tanh, GELU (Hendrycks and Gimpel, 2016), SiLU (Hendrycks and Gimpel, 2016; Ramachandran et al., 2017), and Leaky ReLU (Maas et al., 2013) activations, as well as ReLU activations as baseline comparison. The architecture consists of three fully-connected layers, each with 100 units. For ReLU, sigmoid, Tanh, and Leaky ReLU networks, weight matrices were initialized using Kaiming normal initialization (He et al., 2015). For SiLU and GeLU networks, weight matrices were initialized using Xavier initialization (Glorot and Bengio, 2010), as it is more commonly done in the literature. In either case, bias vectors were initialized to zero. The step-size of SGD was tuned for each learning system.

Performance followed a similar pattern as before: an initial increase followed by a steady decrease. (Figure 3.2d). Different activation functions exhibit different slopes in their decrease in performance. However, all the learning systems experienced a loss of plasticity.

The results in this section demonstrate that a wide range of learning systems based on fully-connected networks exhibit plasticity loss in the Permuted MNIST problem. A natural question is whether plasticity loss is an inherent property of deep learning systems or, if not, whether there is a way to prevent it. Before exploring this question, it is helpful to understand what other phenomena are observed in the network as the learning system loses plasticity.

3.3 Correlates of Loss of Plasticity

The difficulty of training artificial neural networks has been the subject of extensive study in deep learning. There are many different ways in which the training of neural networks can fail. Neurons in the network could become inactive (Shin and Karniadakis, 2020; Lu et al., 2020), the gradients could become too small and uninformative (Bengio et al., 1994), weights could grow exponentially large (Philipp et al., 2017), or the hidden layers in the network could lose representational power

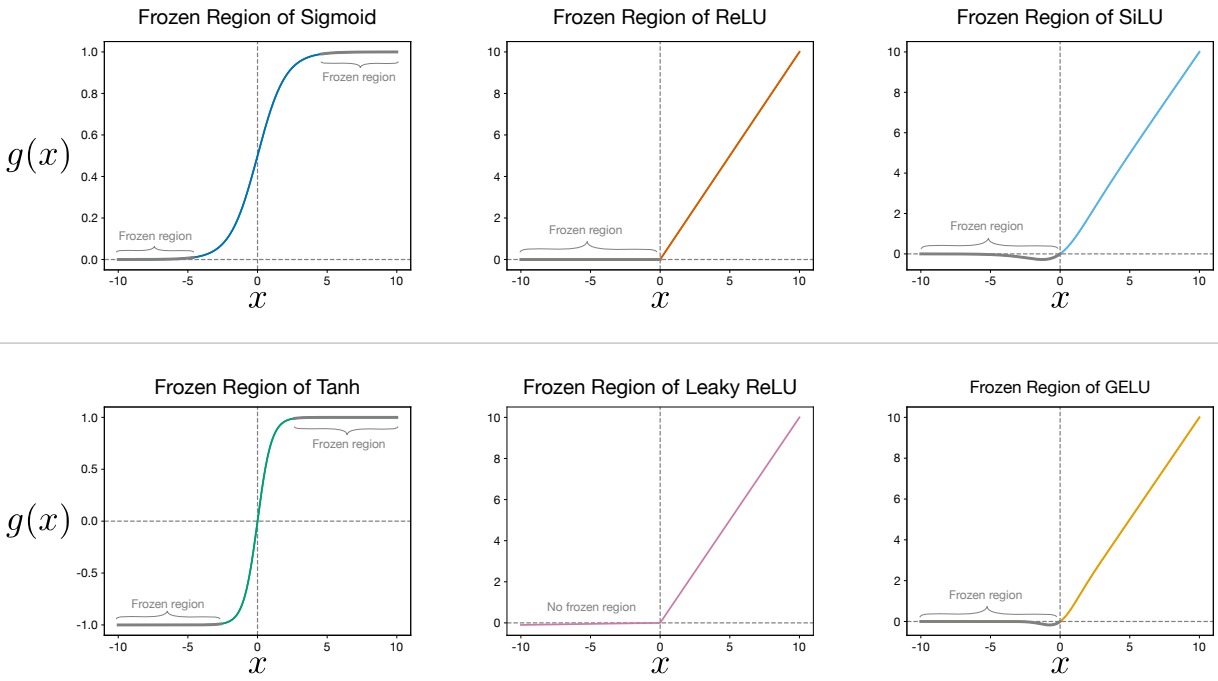

Figure 3.3: Illustration of the frozen region for different activation functions. For Sigmoid and Tanh activations, this illustration uses ϵ of 0.01.

(Kumar et al., 2021). This section explores whether any of these training instabilities are related to the plasticity loss observed in the Permuted MNIST problem. This section focuses on four pathological scenarios: a considerable accumulation of units with constant or nearly constant outputs, an increasing trend in the average magnitude of the weights in the network, a decreasing trend in the average gradient of the loss, and a decrease in the stable rank of the representation layer.

Accumulation of units with constant or nearly constant outputs

In a fully-connected layer, a unit corresponds to the outputs of the nonlinear function in the layer. In other words, a unit represents the operation $g(\mathbf{w} \cdot \mathbf{x} + b)$ for a nonlinear function g, weight vector $\mathbf{w}$, input vector $\mathbf{x}$, and bias term b. The weights of units represent connections that lead to other layers in the network. The output of a unit affects the network's prediction and the gradients flowing back from the unit.

I refer to units with constant or nearly constant values as *frozen* because they see little to no variability in their output. The precise definition of a frozen unit is different depending on the type of activation function. Nevertheless, they all share two key characteristics: they stop contributing to their layer's output, and they tend not to change or change slowly once they become frozen. To determine whether a unit is frozen, the unit's output is measured on a batch of m different

examples, $\mathcal{B} = \{\mathbf{x}_i\}_{i=1}^{m}$, to check whether it is constant or nearly constant. Figure 3.3 gives an illustration of when each different activation function is considered frozen.

ReLU activations are said to be frozen when their output is zero for all observations in $\mathcal{B}$. Specifically, denoting the ReLU function as g, a ReLU unit is considered frozen if $g(\mathbf{w} \cdot \mathbf{x}_i + b) = 0 \ \ \forall \ \ \mathbf{x}_i \in \mathcal{B}$. ReLU units that fall into such a state are also known as dead ReLUs (Shin and Karniadakis, 2020; Lu et al., 2020). A ReLU unit can become permanently inactive when all its inputs are positive and its weights and bias are negative. In such a case, the gradients flowing back from the activation function are always zero, so the unit's input weights remain unchanged during the optimization process.

Sigmoid and Tanh units are classified as frozen when their output is always ϵ-close to the lower or upper bounds of the unit for a small positive ϵ. Specifically, letting a and b be the lower and upper bounds of the activation, respectively, a unit is considered dormant if $g(\mathbf{w} \cdot \mathbf{x}_i + b) < a + \epsilon \ \ \forall \ \ \mathbf{x}_i \in \mathcal{B}$, or if $g(\mathbf{w} \cdot \mathbf{x}_i + b) > b - \epsilon \ \ \forall \ \ \mathbf{x}_i \in \mathcal{B}$. These types of units are also known as saturated (Montavon et al., 2012; Rakitianskaia and Engelbrecht, 2015). When units are saturated, their outputs are almost constant, essentially serving as another bias term with some slight variation. Saturation is a less severe state than death in ReLU activations since it is never irreversible. However, gradients become extremely small at saturation, leading to a significant slowdown in learning.

GELU and SiLU units are an interesting case because, although they output non-constant values on the negative side, their gradients are significantly smaller there than on the positive side (see Figure 3.3). For this reason, GELU and SiLU units are considered frozen when their outputs are always less than or equal to zero. In other words, these units are considered frozen if $g(\mathbf{w} \cdot \mathbf{x}_i + b) \leq 0 \ \ \forall \ \ \mathbf{x}_i \in \mathcal{B}$. Like Tanh and Sigmoid units, the frozen states of GELU and SiLU units are not irreversible. However, gradients of the weights of frozen GELU and SiLU units are severely decreased, and the contribution of units to their corresponding layer is nearly zero. Lastly, there is no notion of a frozen unit for Leaky ReLU activations since they are specifically designed to prevent such a problem.

To measure the percent of frozen units in the Permuted MNIST problem, the activations of units are calculated on a batch of $1,500$ images before the start of each new task. For Sigmoid and Tanh activations, ϵ is set to 0.01. Note that this choice of ϵ implies there are different thresholds for when a unit is considered frozen depending on the specific activation function. However, the precise threshold is less relevant than the trend in the percent of frozen units as the system learns from new tasks.

The question of interest is whether the systems in the previous section accumulate frozen units as they learn from new tasks. Figure 3.4a shows the same results from the previous section next to the percent of frozen units in Figure 3.4b. A clear pattern emerges when comparing the results in the two plots. The decrease in average online accuracy was accompanied by an increase in the

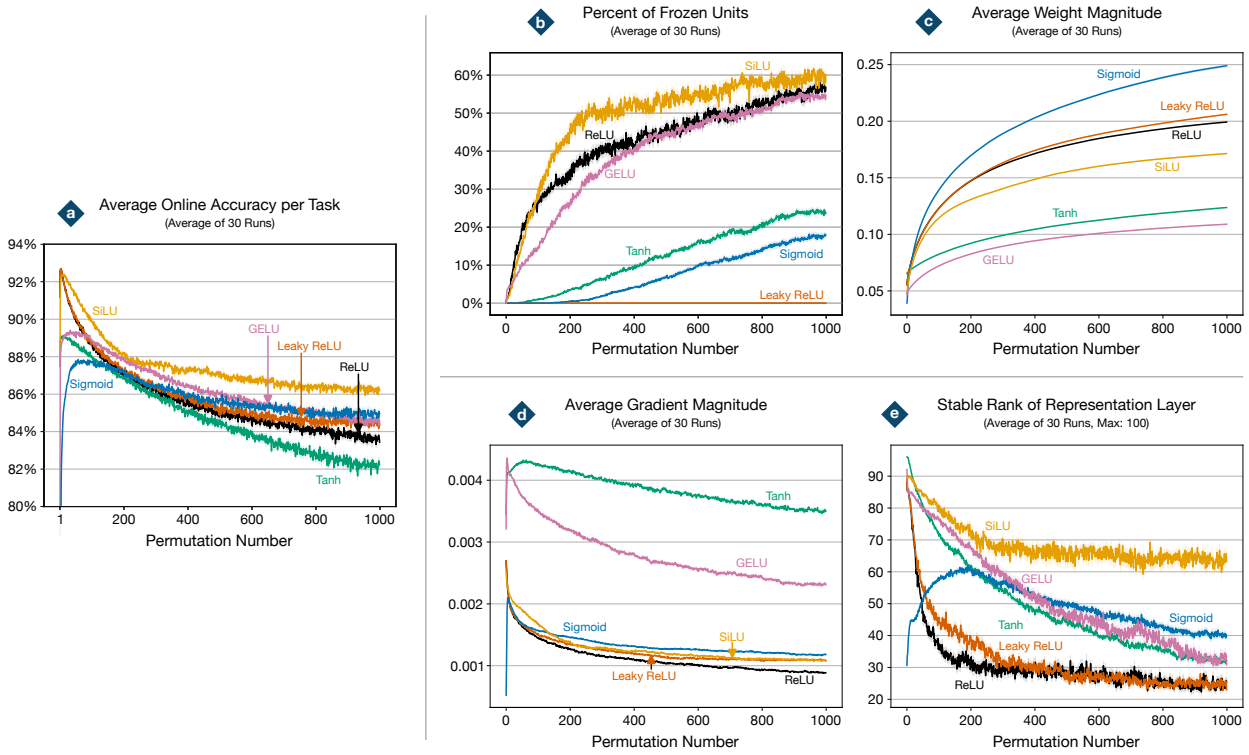

Figure 3.4: **a.** Plasticity loss with different activation functions. **b.** Percent of frozen units. **c.** Average weight magnitude. **d.** Average gradient magnitude. **e.** Stable rank of the representation layer. As performance decreased, the percentage of frozen units and average weight magnitude increased, while the average gradient magnitude and the stable rank of the representation layer decreased. All results are the average of 30 runs. Shaded regions represent one standard error from the average.

percent of frozen units, except for Leaky ReLU activations. Nevertheless, frozen units cannot be considered the cause of plasticity loss since Leaky ReLU networks also experienced plasticity loss despite having no frozen units.

Increasing weight magnitude

Another co-occurring phenomenon to the loss of plasticity is a steady increase in the average magnitude of the network's parameters. Figure 3.4a shows the performance of three-hidden-layer networks with different activations, and Figure 3.4c shows the average weight magnitude of all the parameters in the network measured before the start of each task. As the learning systems lost plasticity, the network's weights also showed an increase in average magnitude.

The increase in weight magnitude could explain the decrease in performance observed in Figure

3.4a. The magnitude of the initial weights of an optimization process is directly linked to the condition number of the Hessian matrix of the loss function with respect to the weights of the network. The condition number of the Hessian is known to affect the speed of convergence of gradient descent algorithms (see Boyd and Vandenberghe, 2004, p. 473 for a clear illustration in convex optimization). Consequently, an increase in the magnitude of the weights could lead to an ill-conditioned Hessian matrix, resulting in a slower convergence. This could explain why networks with Leaky ReLU activations lost plasticity, even when they had no frozen units, as shown in Figure 3.4b.

Decreasing gradient magnitude

Another correlate of plasticity loss in Permuted MNIST is a steady decrease in the magnitude of the gradient of the loss with respect to the network's parameters. Figure 3.4d shows the average gradient magnitude of the parameters of the network computed on each mini-batch and aggregated for each task. As the performance of the different learning systems decreased, the average magnitude of the gradient also decreased. Since the SGD update is proportional to the gradient of the loss function, a smaller gradient magnitude implies slower progress on each optimization step. Thus, a decrease in gradient magnitude could explain the decline in performance resulting from plasticity loss.

Decrease of the stable rank of the representation layer of the network

Unlike the rank of a matrix, which corresponds to the number of linearly independent dimensions, the stable rank considers the contribution of each dimension to the transformation induced by the matrix and only counts the most important ones. A high stable rank indicates that most of the dimensions are relevant to the transformation induced by the matrix. On the other hand, a low stable rank implies that most dimensions have no notable effect on the transformation, implying the information in most dimensions is close to being irrelevant.

The last hidden layer of the network—the *representation layer*—corresponds to the features directly used for producing the output of the network. The stable rank of the representation layer corresponds to the number of essential features required to generate the network's output. To compute the stable rank, the activations of the last layer are calculated using a batch of m observations, $\mathcal{B} = \{\mathbf{x}_i\}_{i=1}^{m}$, resulting in feature vectors $\boldsymbol{\phi}(\mathbf{x}_i) \in \mathbb{R}^d$ for $i \in [m]$, where d is the number of hidden units in the representation layer. The feature vectors are stacked into a matrix, $\boldsymbol{\Phi} \in \mathbb{R}^{m \times d}$, where rows correspond to observations and columns correspond to features. Let σ_i be the singular values of matrix $\boldsymbol{\Phi}$, and assume they are ordered in decreasing value, $\sigma_1 > \sigma_2 > \cdots > \sigma_k$, for $k = \min(m, d)$. The stable rank is defined as the minimum number of largest singular values whose

sum is greater than or equal to $1 - \delta$ of the sum of all singular values (Kumar et al., 2021):

$$\text{srank}_\delta(\mathbf{\Phi}) = \min \left\{ j : \frac{\sum_{i=1}^{j} \sigma_i}{\sum_{i=1}^{k} \sigma_i} \geq 1 - \delta \right\}.$$

In the Permuted MNIST experiment, we compute the stable rank on a batch of 1,500 observations before the start of each task, using a δ value of 0.01.

There was a clear relationship between average online accuracy and a network's stable rank. The decrease in performance in Figure 3.4a was accompanied by a reduction in stable rank in Figure 3.4e. A low stable rank on itself is not necessarily a problem. Past evidence suggests that gradient-based optimization favours low-rank solutions (Yang et al., 2019; Razin and Cohen, 2020; Smith et al., 2021). However, a low-rank representation may not be the best starting point for learning about a new task. A decrease in the stable rank has also been associated with a decreased ability to fit sequences of value functions in reinforcement learning (Kumar et al., 2021).

Takeaways of the correlates of loss of plasticity

This section presented co-occurring phenomena that may explain the loss of plasticity observed in the Permuted MNIST problem. However, we cannot conclude that these phenomena cause plasticity loss based on these observations, as their correlation is tenuous. For example, although SiLU networks in Figure 3.4b had the highest number of frozen units, they also achieved the highest performance for most of the tasks in Figure 3.4a. Sigmoid networks had the highest average weight magnitude on the final tasks in Figure 3.4c; however, they still achieved higher average online accuracy than GELU networks, which had the smallest weight magnitude across all networks. Tanh networks exhibited a high average gradient magnitude throughout the experiment, as shown in Figure 3.4d; yet they also achieved the worst performance on the last 100 tasks. Lastly, Tanh networks had a similar stable rank to GELU networks and a higher stable rank than ReLU networks, as shown in Figure 3.4e, yet they still performed worse throughout the experiment. Thus, although a clear pattern emerges in the results: plasticity loss is accompanied by an increase in frozen units and average weight magnitude, and a decrease in average gradient magnitude and stable rank, none of these phenomena is a perfect predictor of plasticity loss.

Nevertheless, these phenomena correspond to pathological scenarios that can be addressed to prevent plasticity loss. A large number of frozen units and a low stable rank indicate a decrease in the network's representational capacity, thereby affecting its ability to approximate a wide variety of solutions. On the other hand, a high average weight magnitude and a low average gradient magnitude could slow down gradient-based optimization, leading to poor online performance. The following section examines several methods for preventing plasticity loss and explores their impact on the correlates of plasticity loss.

3.4 Mitigating Plasticity Loss

This section presents the results of various modifications to the learning systems based on ReLU networks used in the previous two sections. The goal is to study existing techniques that may mitigate plasticity loss and investigate how they impact its correlates. A secondary goal is to provide a prototype for evaluating algorithms for maintaining plasticity in future chapters. These types of evaluations first define a base learning system, or simply *base system*, and then several modifications to it. In this section, the base system is a fully-connected network with three hidden layers and 100 ReLU units per layer trained with SGD. The modifications added to the base system are: L2 regularization, shrink-and-perturb, concatenated ReLU activations, layer norm, residual connections, and dropout.

L2 regularization constrains the L2-norm of the network's parameters from growing too large. It modifies the SGD update function to

$$\boldsymbol{\theta}_{t+1} = (1 - \lambda\alpha)\boldsymbol{\theta}_t - \alpha\nabla_{\boldsymbol{\theta}_t}\ell\left(f_{\boldsymbol{\theta}_t}(\mathbf{x}_t), \mathbf{y}_t\right),$$

where $\lambda > 0$ is a hyperparameter called regularization factor. In the Permuted MNIST experiment, L2 regularization may help prevent the network's average weight magnitude from growing too large. In the evaluations, L2 regularization employs the same step-size as the base system but uses a tuned regularization factor.

A simple addition to L2 regularization is to include a small amount of parameter noise, resulting in the update:

$$\boldsymbol{\theta}_{t+1} = (1 - \lambda\alpha)\boldsymbol{\theta}_t - \alpha\nabla_{\boldsymbol{\theta}_t}\ell\left(f_{\boldsymbol{\theta}_t}(\mathbf{x}_t), \mathbf{y}_t\right) + \boldsymbol{\epsilon}_t,$$

where $\boldsymbol{\epsilon}_t \sim \mathcal{N}(\mathbf{0}, \sigma^2\boldsymbol{I})$ is a small amount of Gaussian noise added to each parameter of the network, and σ^2 is the variance of the noise and is treated as a hyperparameter. Using a small amount of parameter noise is also a common regularization technique in deep learning (Goodfellow et al., 2016). The combination of L2 regularization and parameter noise is also known as shrink-and-perturb, a term coined by Ash and Adams (2020). In the evaluations in this section, shrink-and-perturb uses the same regularization factor and step-size as L2 regularization, but the noise variance is tuned.

Dropout is a standard regularization technique often used in state-of-the-art applications. While not directly targeting any of the correlates of plasticity loss, it is included here due to the widespread use of the technique. The dropout probability is tuned in the evaluations in this section.

The other three approaches explored in this section are architectural changes proposed to facilitate learning in neural networks. Concatenated ReLU activations, or CReLU, were proposed to prevent plasticity loss in reinforcement learning (Abbas et al., 2023). For an input x, the CReLU

operation is defined as $\text{CReLU} \doteq [\text{ReLU}(x), \text{ReLU}(-x)]$. By design, one of the two outputs of the CReLU activation will have a non-zero gradient, completely removing the problem of dying ReLU units. Another consequence of the CReLU activation is that it doubles the number of hidden units in the layer since each activation now produces two outputs. During evaluations, the optimizer's step-size is tuned for systems with CReLU activations. Moreover, the number of CReLU activations used in each layer is set to 50, so the number of units remains 100 after applying the CReLU operation.

Layer norm is a technique that normalizes the activations of units by their mean and standard deviation. Let $\overline{\text{x}} \doteq \frac{1}{n}\sum_{i=1}^{n}\mathbf{x}[i]$ and $\text{s}^2 = \frac{1}{n}\sum_{i=1}^{n}(\mathbf{x}[i]-\overline{\text{x}})^2$. Layer norm performs the operation

$$\text{Layer Norm}(\mathbf{x}) \doteq \frac{\mathbf{x}-\overline{\text{x}}}{\sqrt{\text{s}^2+\epsilon}} \odot \boldsymbol{\gamma} + \boldsymbol{\beta}, \tag{3.1}$$

where ϵ is a small positive number to prevent division by zero, and $\boldsymbol{\gamma}$ and $\boldsymbol{\beta}$ are learnable parameter vectors. This type of normalization layer has been shown to smooth out the loss landscape, making the optimization more well-behaved (Santurkar et al., 2018) and enabling the use of a higher step-size, which may speed up learning (Bjorck et al., 2018). During evaluation, the step-size of systems using layer norm was tuned. Moreover, two different ways of applying layer norm were tested: before or after the activation function. Applying layer norm after the activation resulted in higher performance, so the results correspond to that setting.

The last architectural modification tested in this section is the use of residual connections. Residual connections are a standard technique in deep learning that provides shorter pathways for gradients to flow to deeper layers (He et al., 2016). Note that new pixel permutations do not make the information stored in the network completely irrelevant. If the same permutation were applied to the columns of the input weight matrix, the network would not experience any drop in performance. This implies that only input weights have to be modified to recover from a random pixel permutation. However, standard fully-connected networks update all weights concurrently. Moreover, layers closer to the input units are often the slowest to change due to gradients shrinking as they flow back through the network (Bengio et al., 1994; Pascanu et al., 2013). Residual connections provide a more direct path for gradients to flow, which could speed up learning and prevent unnecessary changes to other layers. In the experiments, the step-size parameter of systems using residual connections is tuned. Residual connections were implemented by adding the output of the first layer to the outputs of the second and third layers, and the output of the second layer to the output of the third layer.

The goal of this experiment is to determine if any of these modifications help mitigate plasticity loss. A secondary goal is to observe whether mitigating plasticity loss also prevents the four correlates discussed in the previous section.

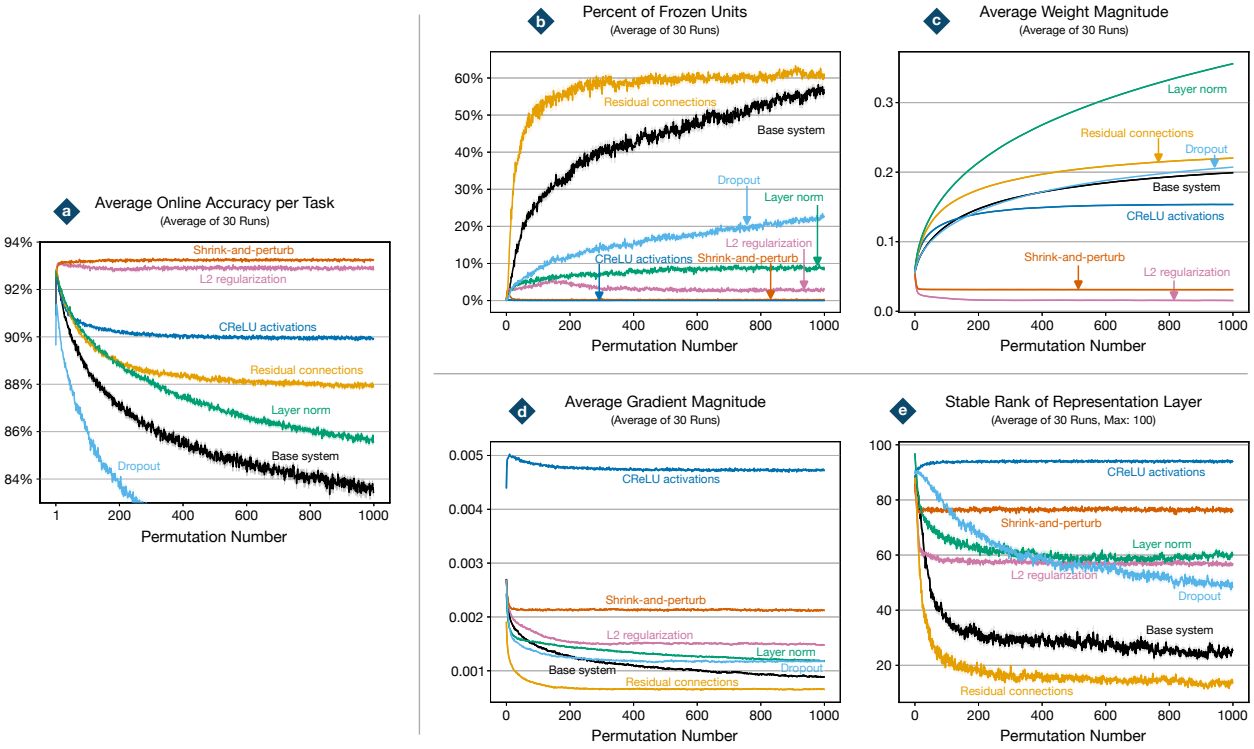

Figure 3.5: **a.** Plasticity loss with different modifications to the base system. **b.** Percent of frozen units. **c.** Average weight magnitude. **d.** Average gradient magnitude. **e.** Stable rank of the representation layer. Except for dropout, all the modifications mitigated plasticity loss to some extent, but only shrink-and-perturb experienced no noticeable drop in performance. All results are the average of 30 runs. Shaded regions represent one standard error from the average. Hyperparameter values are given in Table A.2.

Each modification improved the average online accuracy of the base system. However, most of them still showed some drop in performance, suggesting they did not completely prevent plasticity loss. Figure 3.5a shows the performance of each different system in Permute MNIST, whereas Figures 3.5b to 3.5e show the four measurements corresponding to the correlates of loss of plasticity. Shrink-and-perturb was the only modification that did not have a perceptible drop in performance. Moreover, shrink-and-perturb also prevented the pathological scenarios related to the correlates of loss of plasticity: it did not have any frozen units (Figure 3.5b); it maintained a small average weight magnitude (Figure 3.5c); it did not have a drop in average gradient magnitude (Figure 3.5d); and it maintained a high stable rank (Figure 3.5e).

These results further emphasize that the correlates of plasticity loss are not perfect predictors of the phenomenon. For example, the system with residual connections had a higher percent of frozen units, a higher weight magnitude, a lower gradient magnitude and a lower stable rank than the base system. At the same time, the system with residual connections had better performance

than the base system throughout the entire experiment. The correlates of plasticity loss do not explain this difference in performance.

The most important takeaway of this section is that loss of plasticity is not an inherent aspect of deep learning systems. With appropriate modifications, deep learning systems can be designed to retain plasticity while learning continually. Moreover, by exploring various modifications to ReLU networks, we identified reasonable baselines to compare against in the subsequent chapters of this dissertation.

3.5 History and Other Relevant Work on Plasticity Loss

The study of how learning systems based on neural networks slowly lose their ability to learn has a long history spanning several decades. This section summarizes the literature on plasticity loss and its measurement. The purpose of this section is to provide the reader with additional context on how this chapter contributes to the broader study of plasticity loss in deep learning systems.

History of plasticity loss

The loss of plasticity phenomenon has been observed multiple times and received different names over the last four decades. The earliest study of the loss of the ability to learn I could find also employed the now-popular term plasticity. Munro (1986) studied how a single hidden unit lost its ability to learn after a critical learning period during which the neuron was most plastic. This initial description of the phenomenon is what we now call unit saturation in Sigmoid and Tanh units.

In the early 2000s, the phenomenon was demonstrated in the psychology literature, where it was used as a modeling tool for different psychological effects. Ellis and Lambon Ralph (2000) used neural network systems to model the age-of-acquisition effect, which refers to the tendency for people to recall words learned early in their lives more readily than words learned later in life. Multiple follow-up papers also demonstrated that learning systems based on shallow, fully-connected networks lost their ability to learn from new data after initial training (Smith et al., 2000; Zevin and Seidenberg, 2002; Ralph and Ehsan, 2006; Mermillod et al., 2009). Thomas and Johnson (2006) used neural networks to model sensitive learning periods in humans. Recovery from lesions decreased with training duration, suggesting that sensitive periods correspond to early training stages when networks retain the plasticity to relearn damaged representations. As part of this wave of early studies of plasticity loss, we find the first paper published in a machine learning conference by Smith et al. (2000). However, the phenomenon did not capture the attention of the machine learning community, which devoted greater effort to studying catastrophic forgetting

(McCloskey and Cohen, 1989; French, 1999).

Two decades later, the machine learning literature began to show evidence of the phenomenon, but it did not systematically study it. Chaudhry et al. (2018) demonstrated that neural networks suffer from loss of plasticity, which they termed intransigence, and catastrophic forgetting in class-incremental learning with fully-connected and convolutional networks. Achille et al. (2018) found critical learning periods during which learning in fully-connected and convolutional networks was more sensitive to distortions in the data stream. They observed that the Fisher information of the network's weights decreased after those critical learning periods; thus, they claimed that the network was losing information plasticity. Ash and Adams (2020) demonstrated that fully-connected and convolutional networks lost their ability to learn from new observations if they were initially pre-trained on a smaller subset of the dataset. Both Dohare (2020) and Rahman (2021) noted that features in fully-connected networks lost adaptiveness when trained on sequences of non-stationary learning problems.

The loss of the ability to learn was also noted in the deep reinforcement learning (RL) literature. Igl et al. (2021) observed that the non-stationary observation distribution experienced by deep RL agents adversely affected performance. Kumar et al. (2021) noted that the stable rank of the representation layer of deep RL agents decreased over time. The phenomenon, which they referred to as implicit under-parameterization, was accompanied by an increased inability to fit sequences of value functions. Nikishin et al. (2022) noted that deep RL agents tended to overfit to initial experiences, decreasing their ability to learn from new observations—an effect they called primacy bias. Lyle et al. (2022) also found that deep RL agents lost their ability to update their predictions quickly and referred to the problem as capacity loss. Sokar et al. (2023) also found a decreased ability to learn from new observations in deep RL, which they attributed to an accumulation of dormant neurons, which I call frozen units in this dissertation.

While the number of research papers studying the loss of the ability to learn was rapidly increasing, none of them recognized the effect as the same underlying phenomenon. This changed once my co-authors presented our results from an early manuscript at the Conference on Lifelong Learning Agents (Sutton and Dohare, 2022). In the corresponding manuscript, we used the term loss of plasticity and identified previous work on loss of the ability to learn as the same underlying phenomenon (Dohare et al., 2023). This effort by my co-authors and me resulted in a publication in Nature (Dohare et al., 2024) and was quickly followed by a large number of papers directly studying the phenomenon (Lee et al., 2024; Elsayed et al., 2024; Lyle et al., 2024b; Dohare et al., 2024; Elsayed and Mahmood, 2024; Chung et al., 2024; Farias and Jozefiak, 2025; Lewandowski et al., 2025). Along the way, a wide variety of methods have been proposed to address plasticity loss; however, I defer discussion of these methods to the following chapters.

Numerical definitions of plasticity loss

Throughout this dissertation, I define loss of plasticity as the loss of the ability to learn. However, there are many ways to operationalize this definition and obtain a concrete measurement that we can reason about for scientific research. In my view, there are two desirable traits of a numerical definition for scientific research: (1) it captures the essence of the phenomenon under study, and (2) it is a measurable quantity that researchers can replicate multiple times. With this in mind, let us review the different numerical definitions of plasticity loss found in the literature. To use the same language as Chapter 2, everything is described in terms of the loss, which follows the opposite trend as the accuracy in the results of this chapter. Nevertheless, as long as the measure is a suitable indicator of learning, accuracy and loss can be used interchangeably in this section.

To be more precise, let us recall the definition of average expected loss from the previous chapter. The average expected loss is defined as

$$\mathcal{L}_k(\boldsymbol{\theta}_{T_k}) \doteq \frac{1}{\tau_k} \sum_{j=0}^{\tau_k - 1} \mathbb{E}_{p_k} \left[\ell \left(f_{\boldsymbol{\theta}_{T_k+j}}(\mathbf{X}_{T_k+j}), \mathbf{Y}_{T_k+j} \right) \right], \tag{3.2}$$

where τ_k is the number of observation-target pairs in the task, T_k is the number of steps before the current task, ℓ is a loss function, $(\mathbf{X}, \mathbf{Y}) \sim p_k$, and $\boldsymbol{\theta}_t$ are the iterates obtained using the SGD update. In a continual learning problem where each task is equally difficult, systems that lose plasticity would show an increasing trend in the average expected loss. If tasks are never repeated, the increase in average expected loss could only be attributed to the loss of the ability to learn. Thus, an increasing trend in average expected loss captures the essence of plasticity loss.

Even if it captures the essence of plasticity loss, the average expected loss requires access to p_k, which is not often available. Because it is not easily measured, the average expected loss is an unsuitable numerical definition of plasticity loss. An alternative is to use the average empirical loss given by

$$\hat{\mathcal{L}}_k(\boldsymbol{\theta}_{T_k}) \doteq \frac{1}{\tau_k} \sum_{j=0}^{\tau_k - 1} \frac{1}{m} \sum_{i=1}^{m} \ell \left(f_{\boldsymbol{\theta}_{T_k+j}}(\mathbf{x}_{T_k+j,i}), \mathbf{y}_{T_k+j,i} \right), \tag{3.3}$$

where $\{(\mathbf{x}_{t,i}, \mathbf{y}_{t,i})\}_{i=1}^{m}$ is a mini-batch of data sampled at time t from p_k. The accuracy version of the equation above is what is used to determine plasticity loss in this chapter. In terms of average empirical loss, we say a system is losing plasticity if the average empirical loss increases. The advantage of this numerical definition is that it is readily available as the network learns and, given enough samples, it should capture the essence of plasticity loss. However, this definition is only valid for studying plasticity loss if all the tasks are equally difficult. If each task is more difficult than the previous one, we could observe an increase in the average empirical loss even without loss of plasticity.

To account for task difficulty, we can use a baseline to compare against. In such a case, we would be interested in the quantity:

$$\hat{\mathcal{L}}_k(\boldsymbol{\theta}_{T_k}) - \hat{\mathcal{L}}_k(\boldsymbol{\theta}_0) \tag{3.4}$$

where $\boldsymbol{\theta}_0$ is a freshly initialized set of parameters. In other words, the quantity in (3.4) is the difference in average empirical loss on task k between a network trained incrementally on the previous and current tasks and a network trained from initialization on the current task. Let us refer to this measure as the *average empirical loss with baseline*. When using this measure, we say that a system is losing plasticity if the average empirical loss with baseline is negative. The definition assumes that a freshly initialized learning system has inherent plasticity, which is not an unrealistic assumption. Thus, this definition captures the essence of plasticity loss, and it is a measurable quantity. The main downside of this definition is that it requires the training of an additional learning system. Chaudhry et al. (2018) use this measure in their study of catastrophic forgetting and intransigence in incremental learning.

The two numerical definitions above do not completely isolate loss of plasticity, since there is still the possibility of positive transfer. By positive transfer, I mean that information learned from one task can help with learning future tasks, resulting in improved performance. We see positive transfer in the results in this chapter. For example, in Figure 3.5a, the performances of the different systems always have an initial increase before steadily decreasing. One may be tempted to conclude that the systems are not losing plasticity during this initial performance. However, the system may be losing plasticity while benefiting from positive transfer. The effect of positive transfer may be completely dominating that of plasticity loss, making the latter imperceptible.

To completely isolate the effect of plasticity loss, one could use a probing methodology. This methodology consists of several probing phases where networks are trained on a synthetic task based on observations from the current task. Specifically, at time T, we begin a probing phase, where neural network $f_{\boldsymbol{\theta}_T}$ is trained on a synthetic task based on $(\mathbf{X}, \mathbf{Y}) \sim p_k$. Instead of using the original targets, $\mathbf{Y}$, we create a new set of targets based on a random function of $\mathbf{X}$. For example, we could use $\mathbf{Y}' = a + \sin(10 \cdot f_{\boldsymbol{\beta}}(\mathbf{X}))$, where $a \in \mathbb{R}$ and $f_{\boldsymbol{\beta}}$ is a network with the same functional form as $f_{\boldsymbol{\theta}_T}$ but with randomly initialized parameters $\boldsymbol{\beta}$. The point of such synthetic targets is to create labels that the network has not seen before but that it should be able to represent. We then define a new data-generating process $(\mathbf{X}, \mathbf{Y}') \sim p'_k$ and measure the quantities in (3.3) or in (3.4) using samples from p'_k as the system learns the synthetic task starting from $\boldsymbol{\theta}_T$. After a certain amount of training, the network's parameters are reset to $\boldsymbol{\theta}_T$, and learning on the original task resumes. In this case, we say a network is losing plasticity if we observe a decreasing trend in the average empirical loss in the synthetic task as the amount of training and probing phases increases.

The probing methodology was proposed by Lyle et al. (2022) for the average empirical loss

and by Lyle et al. (2023) for the average empirical loss with baseline. In addition to isolating the effect of plasticity loss, the probing methodology does not require a notion of tasks, making it ideal for problems without a clear task boundary. However, this numerical definition is not without a downside. The design of synthetic tasks may introduce experimenter bias, and training on synthetic tasks adds computational costs to the evaluation process.

Lastly, one may numerically define plasticity in terms of regret. Let $\boldsymbol{\theta}_t^*$ be the parameter vector that minimizes $\mathbb{E}_{p_k}\left[\ell(f_{\boldsymbol{\theta}}(\mathbf{X}_t), \mathbf{Y}_t)\right]$, where $f_{\boldsymbol{\theta}^*}$ has the same functional form as the approximation function of the learning system. Then, the numerical definition of loss of plasticity based on regret is

$$\frac{1}{\tau_k}\sum_{j=1}^{\tau_k}\mathbb{E}_{p_k}\left[\ell(f_{\boldsymbol{\theta}_{T_k+j}}(\mathbf{X}_{T_k+j}), \mathbf{Y}_{T_k+j})\right] - \mathbb{E}_{p_k}\left[\ell(f_{\boldsymbol{\theta}^*_{T_k+j}}(\mathbf{X}_{T_k+j}), \mathbf{Y}_{T_k+j})\right]. \tag{3.5}$$

This measure is analogous to the average expected loss with baseline, where the baseline is the network parameterized by the best possible choice of $\boldsymbol{\theta}$ at every time step. This definition still has the downside of failing to eliminate the effect of positive transfer. Moreover, requiring access to $\boldsymbol{\theta}_t^*$ limits this definition, as it is rarely available. Yet, the definition allows us to reason precisely about plasticity loss in the abstract.

Each numerical definition captures the essence of plasticity loss under different circumstances. Thus, determining which definition is best is less important than ensuring that the appropriate definition is used, given the specifics of the experimental design. In the Permuted MNIST problem, the average empirical loss definition—an increasing trend of the quantity in (3.3)—is appropriate because every task is equally difficult. Although the measure does not isolate the effect of plasticity loss, the number of tasks eventually allows the influence of plasticity loss to dominate that of positive transfer. Moreover, positive transfer highlights the performance gains that can be achieved by overcoming the loss of plasticity.

3.6 Discussion and Conclusion

The experiments in this chapter demonstrate that a wide range of deep learning systems suffer from plasticity loss, establishing the extent of this phenomenon in fully-connected networks. The settings included several popular deep learning methods commonly used in state-of-the-art systems. For example, residual connections, layer norm, dropout, and GELU activations are common components of fully-connected layers in modern large language model systems (Vaswani et al., 2017; Devlin et al., 2019; Brown et al., 2020). This suggests that the fully-connected layers in large language models may be susceptible to plasticity loss. Moreover, although scaling up the number of parameters in the network mitigated the problem, it did not completely remove it. Increasing the number of network parameters may be a poor strategy for counteracting plasticity loss, as it is expensive and

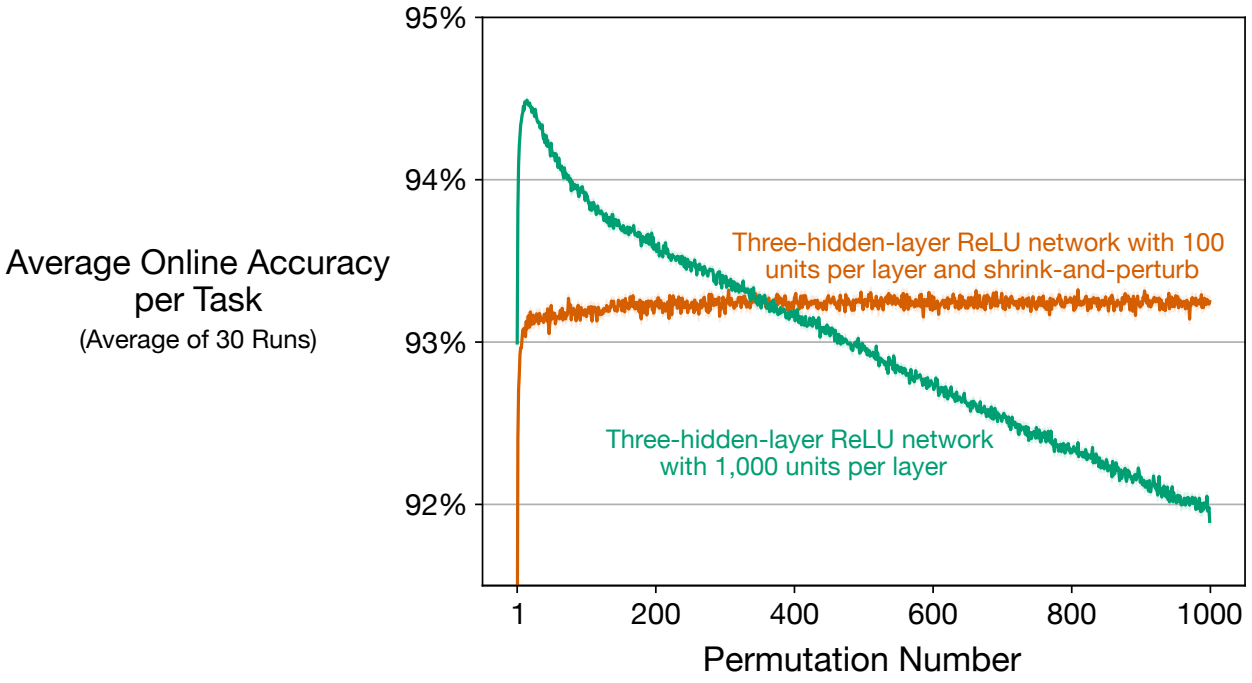

Figure 3.6: Performance of a neural network with 1,000 units per layer and a network with 100 units per layer with shrink-and-perturb. The shrink-and-perturb network eventually surpassed the other network's performance despite being 10 times smaller. The results are an average of 30 runs. Shaded regions represent one standard error from the average.

not entirely effective.

Along with demonstrating loss of plasticity, the chapter also presented co-occurring phenomena that correspond to pathological effects that may slow down learning. However, there is no good argument for any of these correlates having a causal relationship with loss of plasticity. The debate over the causes of plasticity loss is ongoing. For most of the phenomena presented in this chapter, researchers have provided counterexamples where each phenomenon is negatively correlated with plasticity loss (Lyle et al., 2023; Lewandowski et al., 2024). The leading hypothesis at present is that the loss of curvature in the loss landscape has the strongest association with the slowdown in learning (Lyle et al., 2023). However, there is still no strong theoretical evidence to suggest a causal relationship. Nevertheless, the phenomena described in this chapter correspond to extreme pathological scenarios that may suggest different approaches to mitigating plasticity loss or serve as diagnostic tools in live systems.

Finally, the chapter introduced several different approaches from previous work for mitigating plasticity loss. Of all the approaches, shrink-and-perturb was the most successful at preventing plasticity loss. These results are encouraging and suggest that simple and inexpensive approaches

may have a significant impact on the performance of large-scale systems. To illustrate this point, Figure 3.6 shows the performance of shrink-and-perturb with 100 ReLU units and the performance of a network with 1,000 ReLU units. While the shrink-and-perturb network had lower initial performance, it did not decline and eventually surpassed that of a network ten times its size.

Inexpensive and straightforward algorithms, such as shrink-and-perturb, may be essential for further developing the capabilities of large deep learning systems. For example, a recent paper by Springer et al. (2025) found that extended pre-training of language models can negatively impact fine-tuning performance. One possible explanation for this observation is that models lose plasticity during pre-training; if that is the case, an algorithm such as shrink-and-perturb could improve performance by preventing plasticity loss. This explanation becomes more likely given that other work has found that using shrink-and-perturb doubled the convergence speed during fine-tuning of pre-trained models (Verwimp et al., 2025).

A secondary goal of this chapter was to provide evidence for the claim that deep learning systems designed for the stationary framework lose plasticity when trained on non-stationary data. While many of the demonstrations support this claim, a prominent counterexample is the performance of systems trained with L2 regularization. Results with L2 regularization, common in stationary learning systems, showed that it can mitigate plasticity loss in Permuted MNIST. Thus, the results suggest that some techniques designed for the stationary framework may already yield deep learning systems impervious to plasticity loss. While this is true in Permuted MNIST, the results in later chapters will demonstrate that L2 regularization is not always enough to prevent plasticity loss. Throughout the following chapters, I argue that we need to design techniques specifically designed for enabling continual learning. One of such techniques is selective reinitialization, which is introduced in the next chapter.

Chapter 4

Continual Backpropagation and Its Generalization

The results in the previous chapter demonstrated plasticity loss in a wide variety of deep learning systems based on fully-connected layers. Plasticity loss presents a significant challenge for developing deep learning systems capable of continual learning. Fortunately, techniques such as L2 regularization and shrink-and-perturb were capable of preventing the phenomenon, but their effectiveness remains to be established in a broader range of architectures.

This chapter investigates selective reinitialization applied at the unit level as a technique for preventing plasticity loss. The approach is conceptually simple: measure the utility of units in the network and periodically reinitialize those that are no longer useful. To demonstrate the effectiveness of such an approach, I evaluate two algorithms, continual backpropagation (Dohare et al., 2021) and ReDo (Sokar et al., 2023), in the Permuted MNIST problem. Furthermore, I generalize these algorithms into a broader family called *selective unit reinitialization*, which enables systematic examination of how various modifications affect performance.

The results establish selective unit reinitialization as a robust approach for maintaining plasticity. Moreover, I uncover settings where continual backpropagation and Redo fail to maintain plasticity, and address these shortcomings through simple modifications to their utility measures. This chapter presents results from collaborative work published at the 2025 Conference on Lifelong Learning Agents (Hernandez-Garcia et al., 2025), along with additional experiments that deepen the analysis.

4.1 The Importance of Random Initialization and the Benefits of Reinitialization

A crucial step in training deep learning systems occurs before any learning begins: parameter initialization. Initialization involves sampling parameter values from carefully chosen probability distributions. These distributions are designed to enable gradients to propagate through the network without vanishing or exploding, which facilitates initial learning (Glorot and Bengio, 2010; He et al., 2015). The benefits of this random initialization are undone during the learning process. Thus, when learning from a non-stationary data stream, the effects of random initialization are only present early in the learning process.

Even when using a carefully designed initialization function, gradient descent optimization may converge to a suboptimal local minima due to the non-convex nature of the optimization problem (Liu, 2022). An approach for exploring multiple local minima to find the best possible solution is to selectively reinitialize parts of the network. This idea dates back more than 50 years with Selfridge's Pandemonium architecture in 1958. As part of the motivation for the architecture, Selfridge described the same local minima problem encountered nowadays in deep neural networks (see Figure 6 of Selfridge, 1958). The idea of reinitializing units in a nonlinear network of units has been explored in several iterations since then (Klopf and Gose, 1969; Holland and Reitman, 1977; Waugh and Adams, 1995; Kaelbling, 1993; Mahmood and Sutton, 2013; Dohare, 2020; Rahman, 2021).

The motivation for reinitializing units to maintain plasticity is twofold. First, the process of reinitializing units restores some of the initial conditions of the randomly initialized weights that facilitated learning. Second, it enables an additional search process over the parameter space, which may help escape severely suboptimal local minima.

4.2 Selective Reinitialization of Units

Selective reinitialization can be implemented at several levels in a network. This chapter examines the idea of reinitializing units in fully-connected networks. Units in such networks correspond to computational nodes with associated incoming weights, $\mathbf{w} \in \mathbb{R}^d$, bias term, b, and nonlinear activation function, g. Given an input, $\mathbf{x} \in \mathbb{R}^d$, the unit performs the computation $h = g(\mathbf{w} \cdot \mathbf{x} + b)$, which other units can use to build increasingly intricate nonlinear functions. The output h generated by the unit is also called the unit's activation. Reinitializing a unit involves setting new values for its weights, including both incoming and outgoing weights connecting it to other units in the network.

This chapter focuses on two algorithms for reinitializing units: continual backpropagation, or continual backprop for short, and ReDo. Each algorithm was independently proposed, but they

Algorithm 1 Selective unit reinitialization

Input: a fully-connected network
Input: a pruning criterion
Input: a utility measure
Input: a reinitialization method

for each training step **do**
    Receive a mini-batch of data
    Compute predictions of the network and losses
    Compute gradients of the losses with respect to the parameters of the network
    Update parameters of the network using SGD
    for each layer in the network **do**
        Measure the utility of the units in the layer
        Use pruning criterion and utilities to determine how many units to prune
        Set new values for input and output weights of pruned units using reinitialization method

both can be viewed as part of a broader family that I call *selective unit reinitialization* algorithms. This family of algorithms augments learning with backpropagation by periodically executing a reinitialization step. During this step, the algorithm evaluates the utility of units, selects units for pruning, and reinitializes them. More precisely, selective unit reinitialization algorithms have three key components: a *pruning criterion* that determines how many and which units to prune, a *utility measure* which assigns scalar values to each unit and guides the pruning criterion, and the *reinitialization method*, which determines how to assign new values to the weights of new units. The pseudo-code for selective unit reinitialization algorithms is given in Algorithm 1.

ReDo and continual backprop correspond to a specific choice of pruning criterion, utility measure, and reinitialization method. The most significant difference between the two algorithms lies in how they prune units. Thus, let us start by reviewing how each of these algorithms prunes units.

Pruning criteria

The role of a pruning criterion in a selective unit reinitialization algorithm is to determine which units to remove and how many. Specifically, a pruning criterion takes a vector of utilities, $\boldsymbol{u}$, with a real-valued entry for each unit in a layer. Then, based on the utilities, the pruning criterion generates a set of indices, $\boldsymbol{I}$, corresponding to the units to be pruned. The pruning criteria of continual backprop and ReDo include hyperparameters that control the rate at which units are pruned from the network.

The term pruning is not quite accurate, as it implies that units and their connections are permanently removed from the network. Instead, one should think of a pruning criterion as a selection criterion, which chooses units to reinitialize. Nevertheless, these criteria share many

Algorithm 2 Selective unit reinitialization with continual backpropagation pruning

Input: a fully-connected network with L layers parameterized by $\boldsymbol{\theta}_0 = \{\mathbf{W}_{0,0}, \mathbf{W}_{0,1}, \dots, \mathbf{W}_{0,L}\}$
Input: a utility measure
Input: a reinitialization method
Input: maturity threshold M and replacement rate ρ
Initialize: ages, $\boldsymbol{a}_{0,1}, \dots, \boldsymbol{a}_{0,L}$, and cumulative replacement rates, $\kappa_{0,1}, \dots, \kappa_{0,L}$, to zero

for each time step, $t \in \{1, 2, \dots\}$ **do**
  Receive a mini-batch of data, compute predictions and losses
  Compute gradients of the losses with respect to $\boldsymbol{\theta}_{t-1}$ and update using SGD
  for each layer in the network, $l \in \{1, \dots, L\}$ **do**
    Measure the utility of the units in the layer
    Increase ages: $\boldsymbol{a}_{t,l} = \boldsymbol{a}_{t-1,l} + \mathbf{1}$
    Count number of mature units: $m_{t,l} = \sum_{i=1}^{d_l} \mathbb{1}\{\boldsymbol{a}_{t,l}[i] > M\}$
    Accumulate replacement rate: $\kappa_{t,l} = \kappa_{t-1,l} + \rho \cdot m_{t,l}$
    Set $\boldsymbol{I}$ to be an empty list
    if $\kappa_{t,l} \geq 1$ **then**
      Get number of units to prune: $c = \text{Integer Part}(\kappa_{t,l})$
      Get the indices of the c mature units with the lowest utility and store them in $\boldsymbol{I}$
      Decrease cumulative replacement rate: $\kappa_{t,l} = \kappa_{t,l} - c$
      for $i \in \boldsymbol{I}$ **do**
        Set age of new units to zero: $\boldsymbol{a}_{t,l}[i] = 0$
    for $i \in \boldsymbol{I}$ **do**
      Set input, $\mathbf{W}_{t,l-1}[i, :]$ and output, $\mathbf{W}_{t,l}[:, i]$, weights using reinitialization method

similarities with the criteria used in neural network pruning. Thus, I opted to use a slightly imprecise term to preserve the link between the work on selective reinitialization and the neural network pruning literature. Henceforth, pruning a unit indicates that it has been selected for reinitialization.

The pruning criterion in continual backprop selects units in each layer that have low utility at a specific rate. However, units are only eligible for pruning if they have reached a prespecified maturity, measured in parameter updates. Intuitively, new units need time to grow their connections and prove themselves useful before being considered for reinitialization.

In more detail, continual backprop tracks the age of each unit in layer l using a vector, $\boldsymbol{a}_{t,l}$, initialized to zero and incremented by one at each time step $t \in \{1, 2, \dots\}$. Units are protected from pruning until they reach a *maturity threshold*, $M \in \mathbb{N}_0$. The number of mature units in layer l at time t is $m_{t,l} = \sum_{i=1}^{d_l} \mathbb{1}\{\boldsymbol{a}_{t,l}[i] > M\}$, where d_l is the layer size. At each step, continual backprop prunes mature units at a *replacement rate*, $\rho \in [0, 1]$, selecting $\rho \cdot m_{t,l}$ units for reinitialization. Since this quantity may not be an integer, the algorithm accumulates the *cumulative replacement rate*, $\kappa_{t,l} = \kappa_{t-1,l} + \rho \cdot m_{t,l}$. When $\kappa_{t,l} \geq 1$, the algorithms prunes $c = \lfloor \kappa_{t,l} \rfloor$ units with the lowest

Algorithm 3 Selective unit reinitialization with ReDo pruning

Input: a fully-connected network with L layers parameterized by $\boldsymbol{\theta}_0 = \{\mathbf{W}_{0,0}, \mathbf{W}_{0,1}, \ldots, \mathbf{W}_{0,L}\}$
Input: a utility measure
Input: a reinitialization method
Input: reinitialization frequency τ and reinitialization threshold ρ

for each time step, $t \in \{1, 2, \ldots\}$ **do**
  Observe a mini-batch of data, compute predictions, and observe losses
  Compute gradients of the losses with respect to $\boldsymbol{\theta}_{t-1}$ and update using SGD
  for each layer in the network, $l \in \{1, \ldots, L\}$ **do**
    Measure the utility of the units in the layer
    Set $\boldsymbol{I}$ to be an empty list
    if t is a multiple of τ **then**
      Compute average utility: $\overline{u_l} = \frac{1}{d_l} \sum_{i=1}^{d_l} \boldsymbol{u}_l[i]$
      Get the indices of units whose utility is less than or equal to the threshold:
      $$\boldsymbol{I} = \{i : \boldsymbol{u}_l[i] \leq \rho \cdot \overline{u_l}\}$$
    for $i \in \boldsymbol{I}$ **do**
      Set input, $\mathbf{W}_{t,l-1}[i, :]$ and output, $\mathbf{W}_{t,l}[:, i]$, weights using reinitialization method

utilities and decreases $\kappa_{t,l}$ by c. The pruning criterion returns the indices, $\boldsymbol{I}$, corresponding to the low-utility units. The procedure is described in Algorithm 2, where bias terms are omitted for simplicity. $\mathbf{W}_{t,l} \in \mathbb{R}^{d_{l+1} \times d_l}$ correspond to the output weights of layer l, d_l is the number of units in layer l, and $\boldsymbol{\theta}_t$ denotes the set of parameters of the network that are updated using stochastic gradient descent (SGD) on each time step t. The blue lines in the algorithm correspond to the pruning criterion of continual backprop.

ReDo exemplifies a different approach to pruning units by selecting units whose utility falls below a specific threshold at a prespecified frequency. Using the same notation as before, let $\boldsymbol{u}_l$ be the utilities of the units in layer l. ReDo prunes units every certain number of parameter updates, which is a hyperparameter called *reinitialization frequency* and denoted by τ. During pruning, ReDo calculates the average utility of the units in the layer, $\overline{u_l} = \frac{1}{d_l} \sum_{i=1}^{d_l} \boldsymbol{u}_l[i]$. Then, it prunes any unit whose utility is less than or equal to a fraction of the average utility, $\rho \cdot \overline{u_l}$, where $\rho \geq 0$ is a hyperparameter called *reinitialization threshold*. The pruning criterion of ReDo generates a list of indices, $\boldsymbol{I} = \{i : \boldsymbol{u}_l[i] \leq \rho \cdot \overline{u_l}\}$. Pseudo-code for ReDo is given in Algorithm 3, where bias vectors are omitted for simplicity. The blue lines in the algorithm correspond to ReDo's pruning criterion.

The similarities and differences between the pruning criteria of continual backprop and ReDo become apparent when comparing Algorithms 2 and 3. Both criteria allow new units to undergo a certain number of updates before being considered for reinitialization, continual backprop through its maturity threshold, and ReDo through its reinitialization frequency. Similarly, both have a hyperparameter that modulates the number of units pruned on each reinitialization step—continual

backprop's replacement rate and ReDo's reinitialization threshold. The main difference between the two pruning criteria is in the number of units they prune per reinitialization step. Continual backprop prunes at most a fraction of the units in the layer proportional to the replacement rate, whereas ReDo could potentially prune all but one unit if there exists an outlier that disproportionately influences the sample average of the utility of the units, $\overline{u_l}$. On the other hand, the pruning criterion of continual backprop has no stopping condition, so it will continue pruning forever. In contrast, ReDo may stop pruning units if all their utilities surpass $\rho \cdot \overline{u_{t,l}}$ as long as $\rho < 1$. In other words, ReDo's pruning criterion is affected by the distribution of the utilities of the units, whereas continual backprop's pruning criterion is not.

Utility measures

The pruning criterion of selective unit reinitialization is guided by a utility measure that quantifies the importance of each unit. ReDo's utility measure is the mean absolute value of the unit's activation. Formally, letting $(\mathbf{X}, \mathbf{Y}) \sim p$ be observations and labels sampled from the data-generating distribution, p, and $\boldsymbol{h}_l$ be the activations units in layer l. The utility of units is given by

$$\boldsymbol{u}_l = \mathbb{E}_p \left[|\boldsymbol{h}_l| \right].$$

Note that activations are all a function of the observation $\mathbf{X}$. In cases where access to the full distribution of observations and labels is not available, as is the case in most applications, the utility can be estimated from a sample of m observations:

$$\boldsymbol{u}_l = \frac{1}{m} \sum_{t=1}^{m} |\boldsymbol{h}_{t,l}|, \tag{4.1}$$

where $\boldsymbol{h}_{t,l}$ is the activation when processing observation $\mathbf{x}_t$. Alternatively, the utility can also be estimated using a moving average of the most recent time steps:

$$\boldsymbol{u}_{t,l} = \eta \cdot \boldsymbol{u}_{t-1,l} + (1 - \eta) \cdot |\boldsymbol{h}_{t,l}|$$

where $\eta \in [0, 1]$. The experiments in this dissertation only include results using utilities estimated from a mini-batch, as in (4.1). Henceforth, let us refer to the utility in (4.1) as *activation utility*.

The latest version of continual backprop also uses the activation of units, but weighted by the sum of the absolute values of its outgoing weights (Dohare et al., 2024). The intuition is that a unit may still have a significant contribution even when its activation is low if its outgoing weights

are high. For unit i in layer l, the *contribution utility* of the unit is

$$\boldsymbol{u}_l[i] = \mathbb{E}_p\left[|\boldsymbol{h}_l[i]|\right] \sum_{j=1}^{d_l} |\mathbf{W}_l[j,i]|\,,$$

where $\mathbf{W}_l$ is the matrix of outgoing weights. The estimate of this utility measure is given by:

$$\boldsymbol{u}_l[i] = \frac{1}{m}\sum_{t=1}^{m} |\boldsymbol{h}_{t,l}[i]| \cdot \sum_{j=1}^{d_l} |\mathbf{W}_l[j,i]|\,. \tag{4.2}$$

There are many other possible ways to measure the utility of units in a neural network. This chapter also examines two other utility measures commonly used in the pruning literature (Blalock et al., 2020). The first one is the sum of the absolute values of the outgoing weights. The intuition is that the sum of the absolute values of the weights corresponds to how much each unit contributes to the output of the layer, but without accounting for the unit's activation. The utility of unit i is given by:

$$\boldsymbol{u}_l[i] = \sum_{j=1}^{d_{l+1}} |\mathbf{W}_l[j,i]|\,. \tag{4.3}$$

Let us refer to this utility as the *weight magnitude utility*.

Another common utility measure used in the pruning literature is based on a first-order approximation of how much the loss changes when a unit is omitted from the network. Let $\ell(f_{\boldsymbol{\theta}}(\mathbf{x}), \mathbf{y})$ be the loss evaluated on example $\mathbf{x}$ and label $\mathbf{y}$, and $\ell\big(f_{\boldsymbol{\theta}}(\mathbf{x}), \mathbf{y} \mid \boldsymbol{h}_l[i] = 0\big)$ be the loss function when the activation of unit i in layer l is set to zero. To be concise, let us denote $f_{\boldsymbol{\theta}}(\mathbf{x})$ as $\hat{\mathbf{y}}$. The difference in loss when omitting unit i is given by

$$\ell\big(\hat{\mathbf{y}}, \mathbf{y} \mid \boldsymbol{h}_l[i] = 0\big) - \ell(\hat{\mathbf{y}}, \mathbf{y}). \tag{4.4}$$

The difference in (4.4) measures how much the loss changes when unit i is excluded. If the difference is zero, it implies that the unit is irrelevant for making predictions about the given observation. Thus, the difference in (4.4) can be used as a measure of utility. However, computing $\ell\big(\hat{\mathbf{y}}, \mathbf{y} \mid \boldsymbol{h}_l[i] = 0\big)$ for each unit in the network can be prohibitively expensive, so it is often approximated. The first term in (4.4) above can be approximated by a first-order Taylor approximation at the point $\boldsymbol{h}_l[i] = 0$:

$$\ell\big(\hat{\mathbf{y}}, \mathbf{y} \mid \boldsymbol{h}_l[i] = 0\big) \approx \ell(\hat{\mathbf{y}}, \mathbf{y}) + \left(\frac{\partial \ell(\hat{\mathbf{y}}, \mathbf{y})}{\partial \boldsymbol{h}_l[i]}\right)(0 - \boldsymbol{h}_l[i]),$$

where $\frac{\partial \ell\left(\hat{\mathbf{y}}, \mathbf{y}\right)}{\partial \boldsymbol{h}_l[i]}$ is the derivative of the loss with respect to the i-th activation in layer l. The difference

in loss in (4.4) can be then approximated as:

$$\begin{aligned}\ell(\hat{\mathbf{y}}, \mathbf{y} \mid \boldsymbol{h}_l[i] = 0) - \ell(\hat{\mathbf{y}}, \mathbf{y}) &\approx \ell(\hat{\mathbf{y}}, \mathbf{y}) - \Big(\frac{\partial \ell(\hat{\mathbf{y}}, \mathbf{y})}{\partial \boldsymbol{h}_l[i]}\Big)\boldsymbol{h}_l[i] - \ell(\hat{\mathbf{y}}, \mathbf{y}) \\ &= -\boldsymbol{h}_l[i]\frac{\partial \ell(\hat{\mathbf{y}}, \mathbf{y})}{\partial \boldsymbol{h}_l[i]} \\ &= -\sum_{j=1}^{d_{l+1}} \mathbf{W}_l[j, i]\frac{\partial \ell(\hat{\mathbf{y}}, \mathbf{y})}{\partial \mathbf{W}_l[j, i]} \end{aligned} \tag{4.5}$$

For an extended derivation, see Appendix B. The equation above is the difference for a single observation and label. The quantity of interest is the expected difference with respect to the data-generating distribution, p. Moreover, only those units that have no impact on the loss should be considered for pruning. Thus, the utility of a unit is defined as:

$$\boldsymbol{u}_l[i] = \left| \sum_{j=1}^{d_{l+1}} \mathbf{W}_l[j, i] \; \mathbb{E}_p \left[\frac{\partial \ell(\hat{\mathbf{y}}, \mathbf{y})}{\partial \mathbf{W}_l[j, i]} \right] \right|,$$

where the expectation can be estimated using a mini-batch of data:

$$\boldsymbol{u}_l[i] = \left| \sum_{j=1}^{d_{l+1}} \mathbf{W}_l[j, i] \; \frac{1}{m} \sum_{t=1}^{m} \frac{\partial \ell(\hat{\mathbf{y}}_t, \mathbf{y}_t)}{\partial \mathbf{W}_l[j, i]} \right|, \tag{4.6}$$

where $\hat{\mathbf{y}}_t$ and $\mathbf{y}_t$ are the predictions and the target for the t-th sample. Let us refer to the utility measure in (4.6) as *first-order utility*.

Reinitialization methods

The last part that needs to be defined to have a concrete selective reinitialization algorithm for units is the reinitialization method. The standard approach in both ReDo and continual backprop is to reinitialize the unit's incoming weights by sampling new values from the initialization distribution and to set the unit's outgoing weights to zero. In other words, when reinitialization unit i in layer l, new values are sampled from the initialization distribution for each entry j in $\mathbf{W}_{l-1}[i, j]$, and each entry k in $\mathbf{W}_l[k, i]$ is set to zero. Moreover, the input bias term is set to zero, since it corresponds solely to the unit being reinitialized, i.e., $\boldsymbol{b}_{l-1}$ is set to zero, whereas the output bias term is left unchanged, since it still interacts with the values of other units. Generating new input weights injects randomness into the network and serves as a form of random search over the loss landscape. On the other hand, setting the output weights of the new unit to zero prevents it from negatively impacting the loss. Let us call this type of reinitialization *hemi-reinitialization* since only half of the connections of the unit are reinitialized.

Instead of setting the output weights of new units to zero, their value could be sampled from the initialization distribution, just like the input weights. While this approach may initially worsen the loss after reinitialization, it could ultimately speed up how quickly the new unit learns. The gradients of the input weights of a unit are a factor of the output weights of the unit. Consequently, setting the output weights to zero implies that the first few updates to the input weights will be close to zero. Thus, although random reinitialization may be immediately detrimental to the loss, the increased training speed may offset the initial increase in loss. Let us refer to this type of reinitialization as *full reinitialization* since all the values of the weights are reinitialized.

4.3 Effectiveness of Reinitialization Units for Maintaining Plasticity

Given a utility measure, a pruning criterion, and a reinitialization method, it is possible to define an instance of selective unit reinitialization fully. This section evaluates the effectiveness of selective unit reinitialization across a wide variety of deep learning systems. These evaluations are all done in the Permuted MNIST problem introduced in the previous chapter. Each experiment involves a base learning system and several algorithms that build upon it. For any algorithm that builds on the base system, only the additional hyperparameters are tuned. Hyperparameter tuning followed the same procedure as described in the previous chapter: a wide search with a single run followed by a narrower search with multiple runs. For consistency, the learning curves of base systems are coloured black in all the plots.

The first set of experiments focuses on evaluating the performance of the original settings of ReDo and continual backprop. The goal of these evaluations is to examine how selective unit reinitialization performs across different learning systems based on fully-connected networks. The latest version of continual backprop uses the pruning criterion described in Algorithm 2 with contribution utility (equation 4.2) and hemi-reinitialization. On the other hand, the original version of ReDo uses the pruning criterion described in Algorithm 3 with activation utility (equation 4.1) and hemi-reinitialization.

The first evaluation examines how selective unit reinitialization algorithms perform in networks of varying sizes. The base learning systems are ReLU networks with three hidden layers and 10, 100, or $1,000$ units per layer trained using SGD. The mini-batch size of this and all the learning systems described henceforth in the chapter is set to 30. The base system is augmented with continual backprop and ReDo, as well as L2 regularization and shrink-and-perturb for baseline comparisons. The hyperparameter tuned for the base system was the step-size of the optimizer; for continual backprop, the maturity threshold and replacement rate; for ReDo, the reinitialization frequency and threshold; for L2 regularization, the regularization factor; and for shrink-and-perturb, the variance

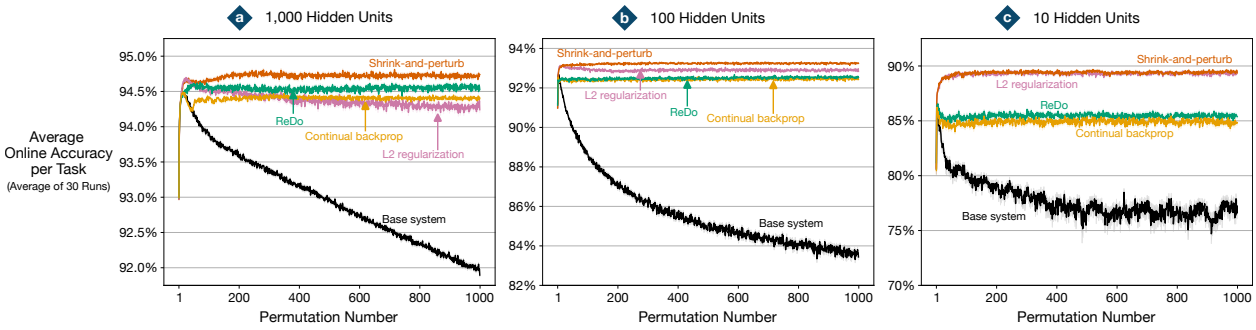

Figure 4.1: **a.** Performance of ReLU networks with three hidden layers with 1,000 units per layer. **b.** Performance of networks with 100 units per layer. **c.** Performance of networks with 10 units per layer. Each line represents the average of 30 independent runs, and the shaded regions indicate one standard error away from the sample average. Hyperparameter values are given in Table A.3.

of the noise, and it used the same regularization factor as for L2 regularization. For more details on hyperparameter tuning, see Appendix A.

Continual backprop and ReDo behaved differently depending on the size of the network (Figure 4.1). When using a network with 1,000 units per layer (Figure 4.1a), ReDo had stable performance, as well as continual backprop after some initial fluctuation. L2 regularization exhibited a steady decline in performance, indicating a sustained loss of plasticity throughout the experiment. In contrast, shrink-and-perturb had stable performance and the highest among the learning systems. In networks with 100 units (Figure 4.1b), shrink-and-perturb, ReDo, and continual backprop had stable performance. In contrast, L2 regularization had only a slight decrease in performance, which stabilized after 200 permutations. Lastly, in networks with 10 units (Figure 4.1c), ReDo and continual backprop had steady performance after an initial decrease. L2 regularization, unlike in the previous two cases, exhibited steady performance, matching that of shrink-and-perturb.

Notably, the gap in performance between shrink-and-perturb and the two selective unit reinitialization algorithms increased as the number of units decreased. This observation suggests that the effectiveness of selective unit reinitialization algorithms was reduced when the network had a small number of units. Interestingly, the opposite effect is observed with L2 regularization: its effectiveness in maintaining plasticity decreased as the number of units increased.

The networks in the previous experiments were all trained with SGD. Next, we examine the effectiveness of selective unit reinitialization algorithms when using optimizers such as SGD with momentum and Adam. Both of these optimizers maintain moving averages of the gradients of the loss, which are used to update the network's parameters. One potential issue with reinitializing weights in the network when using optimizers that store gradient statistics is that freshly reinitialized weights have no relationship to the stored information. Thus, the information stored by the

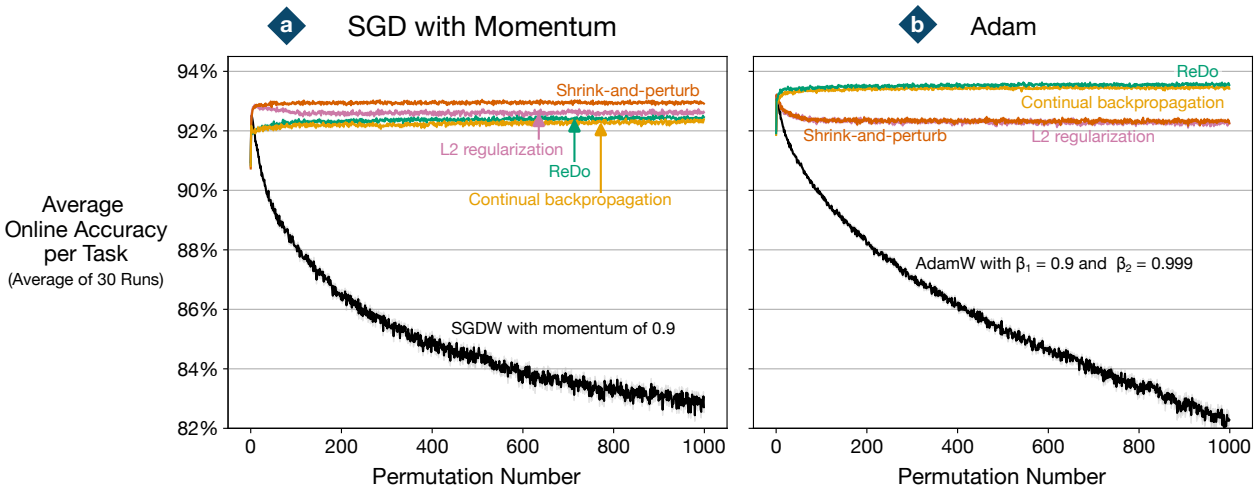

Figure 4.2: **a.** Performance of three-hidden-layer ReLU networks with 100 units per layer trained using SGD with momentum. **b.** Performance of networks trained using Adam. For L2 regularization and shrink-and-perturb, the learning systems used SGDW with momentum and AdamW to prevent learning instabilities unrelated to plasticity loss. Each line represents the average of 30 independent runs, and the shaded regions indicate one standard error away from the sample average. Hyperparameter values are provided in Table A.4.

optimizer may interfere with learning about reinitialized weights, resulting in poor performance.

The base systems in this new set of experiments are three-hidden-layer ReLU networks with 100 units per layer trained with SGD with momentum or Adam. These base systems are augmented with continual backprop, ReDo, L2 regularization, or shrink-and-perturb to test how each algorithm affects plasticity loss. For these experiments, the step-size for both base systems, the momentum term for SGD, and the moving average factors in Adam, β_1 and β_2, were tuned. L2 regularization and shrink-and-perturb utilize SGDW with momentum and AdamW, which differ from the original optimizers when regularization is applied and have been shown to be more stable (Loshchilov and Hutter, 2019).

ReDo and continual backprop had stable performance with both optimizers, suggesting the corresponding learning systems maintained plasticity (Figure 4.2). When using SGD with momentum, shrink-and-perturb and L2 regularization had increased performance over ReDo and continual backprop. However, when using AdamW, L2 regularization and shrink-and-perturb had decreasing performance, suggesting a loss of plasticity. These results suggest that selectively reinitializing units can maintain plasticity with optimizers that use gradient statistics to perform parameter updates. Moreover, the results show that L2 regularization and shrink-and-perturb are not always reliable for maintaining plasticity.

All the learning systems so far use ReLU activations. The following experiments evaluate the

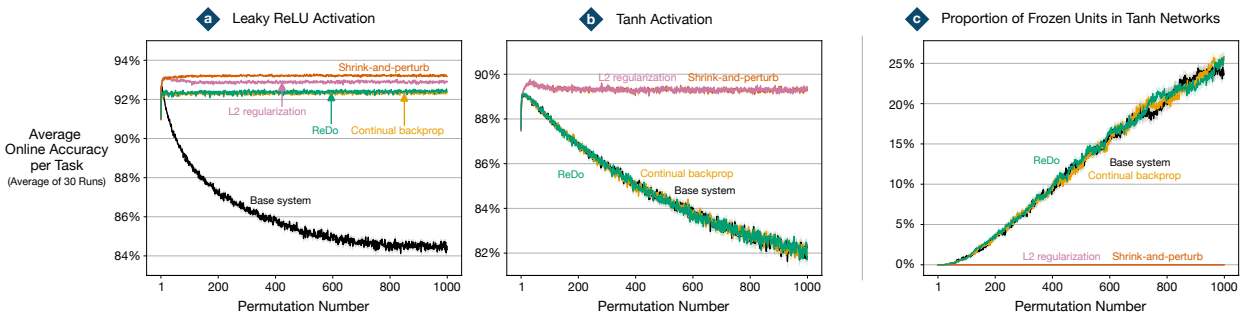

Figure 4.3: **a.** Performance of three-hidden-layer networks with 100 Leaky ReLU units per layer trained using SGD. **b.** Performance of three-hidden-layer networks with 100 Tanh units per layer trained using SGD. **c.** Proportion of frozen units in Tanh networks. Each line represents the average of 30 independent runs, and the shaded regions indicate one standard error away from the average. Hyperparameter values are provided in Table A.6.

performance of continual backprop and ReDo in systems using a variety of activation functions. To reduce the hyperparameter tuning required for these experiments, this evaluation considers only a smaller set of activations from the previous chapter. Both Tanh and Sigmoid activations are S-shaped and may suffer from saturation at either of the extremes of the function. Because their shapes and behaviours are similar, this evaluation considers only Tanh activations and omits Sigmoid activations. GELU and SiLU activations are both smooth versions of ReLU, so continual backprop and ReDO would likely be as effective with them. Therefore, GELU and SiLU activations are excluded from this evaluation. Lastly, Leaky ReLU activations are similar to ReLU, except that they still activate when the input to the function is negative, which completely prevents the problem of dying ReLUs. Since Leaky ReLU is unlike any of the other five activation functions, it is included in this section. Thus, the two base systems are three-hidden-layer networks with 100 Tanh or Leaky ReLU units per layer trained with SGD. For each base system, the step-size parameter was tuned. For all the other algorithms, hyperparameters were tuned as described before.

In the case of Leaky ReLU networks (Figure 4.3a), the performances of the learning systems were similar to systems using ReLU activations (Figure 4.1b). However, in the case of Tanh networks, continual backprop and ReDo had the same performance as the corresponding base system, implying that they failed to maintain plasticity (Figure 4.3b). On the other hand, L2 regularization had stable performance after some initial fluctuation. Finally, shrink-and-perturb behaved similarly to L2 regularization, suggesting that adding parameter noise had a minimal effect on the performance of Tanh networks.

The results with Tanh activations can be explained by the utility measures used in continual backprop and ReDo. Activation utility corresponds to the average absolute value of the unit's activation. As shown in the previous chapter, a large accumulation of frozen units is correlated

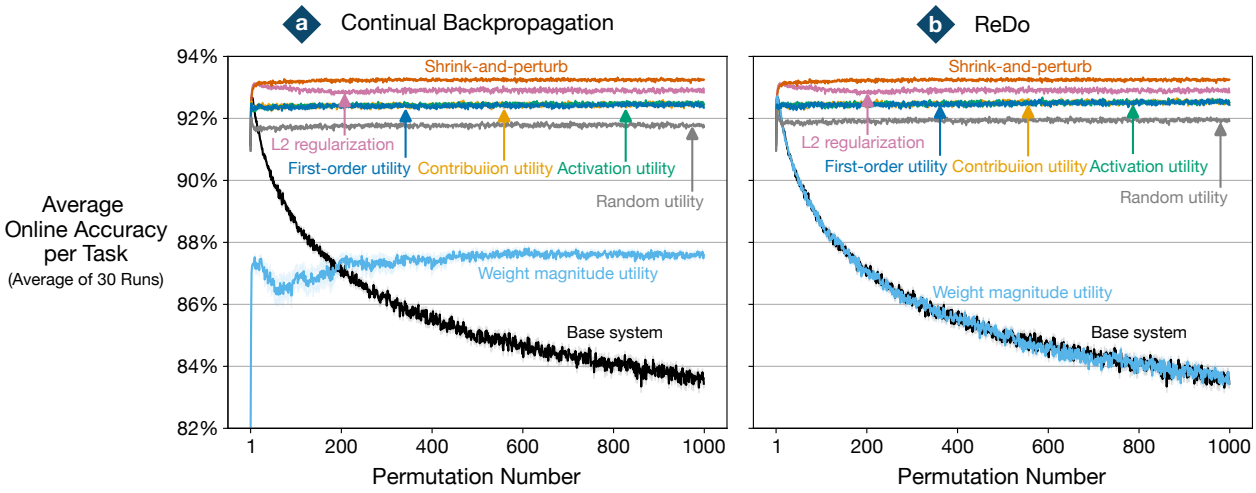

Figure 4.4: **a.** Performance of continual backprop with different utility measures. **b.** Performance of ReDo with different utility measures. Each line represents the average of 30 independent runs, and the shaded regions indicate one standard error away from the average. Hyperparameter values are provided in Table A.5.

with loss of plasticity. In Tanh activations, a unit freezes when its output approaches ± 1. Thus, when a unit is frozen, activation utility assigns it the highest possible utility, thereby incentivizing the preservation of frozen units. Contribution utility may also have a similar effect, since it works in a manner similar to activation utility. Figure 4.3c shows that continual backprop and ReDo accumulate frozen units at the same rate as the base system, suggesting that the algorithms are failing to reinitialize frozen units. This problem could be addressed by considering other utility measures.

The effect of utility measures when selectively reinitializing units

This section studies how utility measures affect the performance of selective unit reinitialization algorithms. For this study, continual backprop and ReDo are modified to use each of the utility measures introduced in Section 4.2. The reinitialization method is the same as in the previous experiments, hemi-reinitialization.

The first experiment in this section examines how the utility measure affects the performance of learning systems using ReLU networks. The base system is a three-hidden-layer ReLU network trained using SGD. The hyperparameters of continual backprop and ReDo were tuned for each different utility measure. Additionally, a utility measure that assigns random values sampled uniformly from $[0, 1]$ is included as a baseline.

The performance of continual backprop or ReDo did not vary much when using first-order,

contribution, or activation utility (Figures 4.4a and 4.4b, respectively). Surprisingly, random utility resulted not only in stable performance but also almost matched the performance of activation, first-order, and contribution utilities. This result suggests that randomly reinitializing units is enough for maintaining plasticity in the permuted MNIST problem. On the other hand, weight magnitude utility resulted in the lowest performance among all the utility measures in both ReDo and continual backprop. In the case of continual backprop, using weight magnitude utility resulted in stable performance after some initial fluctuation. In the case of ReDo, using weight magnitude utility resulted in the same performance as the base system, suggesting it failed to maintain plasticity. Based on these results, it is tempting to suggest that there are no differences between activation, contribution, and first-order utilities and that they may be used interchangeably. However, it is important to note that the behaviours of these three utility measures depend on the type of activation function used in the network. For example, contribution and activation utilities were as effective in Leaky ReLU networks (Figure 4.3a) as in ReLU networks (Figure 4.1b), but performed poorly in Tanh networks (Figure 4.3b).

To verify whether utility measures yield different performance depending on the type of activation function, the following experiment uses networks with Tanh or Leaky ReLU activations. The base systems are a network with Tanh activations and a network with Leaky ReLU activations. In both base systems, the network consists of three hidden layers, each with 100 units, and is trained using SGD. For this experiment, ReDo uses activation or first-order utility, whereas continual backprop uses contribution or first-order utility. Hyperparameters were tuned as described before.

In networks with Leaky ReLU activations, first-order utility resulted in stable performance after some initial fluctuation in both ReDo and continual backprop (Figure 4.5a). In both cases, first-order utility resulted in slightly lower performance. In either case, ReDo and continual backprop performed worse than L2 regularization and shrink-and-perturb. In contrast, first-order utility led to a drastic improvement in performance over activation or contribution utility in Tanh networks. In Tanh networks, ReDo with activation utility and continual backprop with contribution utility resulted in the same performance as the base system; in other words, the learning systems failed to maintain plasticity. On the other hand, first-order utility yielded stable performance with ReDo and continual backprop, suggesting that the learning systems successfully maintained plasticity. Moreover, ReDo's performance matched that of L2 regularization and shrink-and-perturb when using first-order utility, whereas continual backprop surpassed both when using first-order utility. These results suggest that utility measures interact differently with activation functions. Practitioners should carefully consider utility measures based on the specifics of their network architecture.

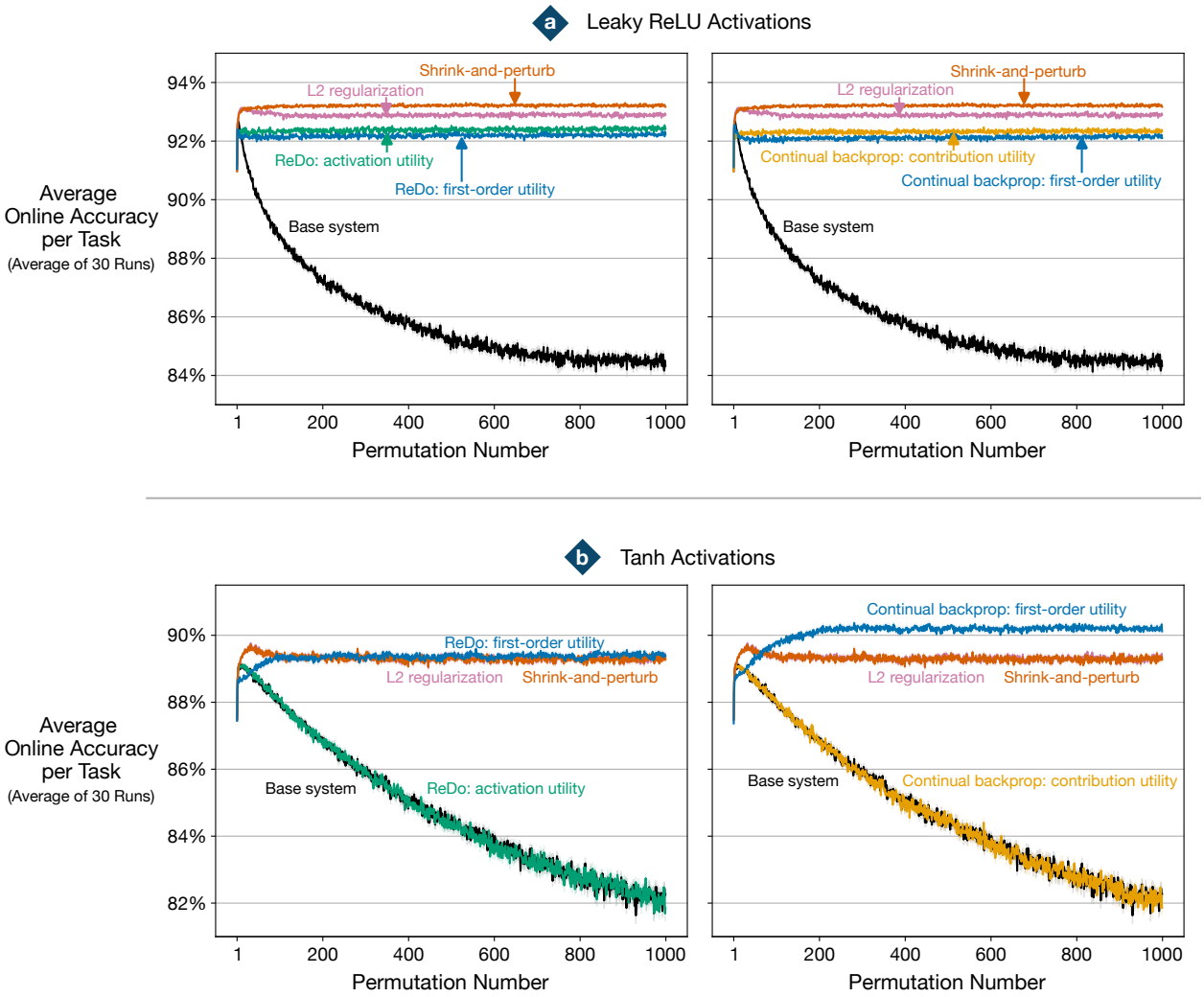

Figure 4.5: **a.** Performance of reinitialization algorithms in networks with leaky ReLU activations. **b.** Performance of reinitialization algorithms in networks with Tanh activations. Each line represents the average of 30 independent runs, and the shaded regions indicate one standard error away from the average. Hyperparameter values are provided in Table A.6.

The Effect of Reinitialization Methods When Selectively Reinitializing Units

In all the previous experiments, ReDo and continual backprop used hemi-reinitialization. New, randomly initialized units are likely to affect the network's output in unexpected ways, so setting their output weights to zero may mitigate their influence. On the other hand, randomization is a key aspect of selective reinitialization algorithms, allowing them to search the parameter space for better solutions. Thus, reducing the amount of randomization may negatively impact the search process. Moreover, setting the output weights of units to zero may slow down the learning process.

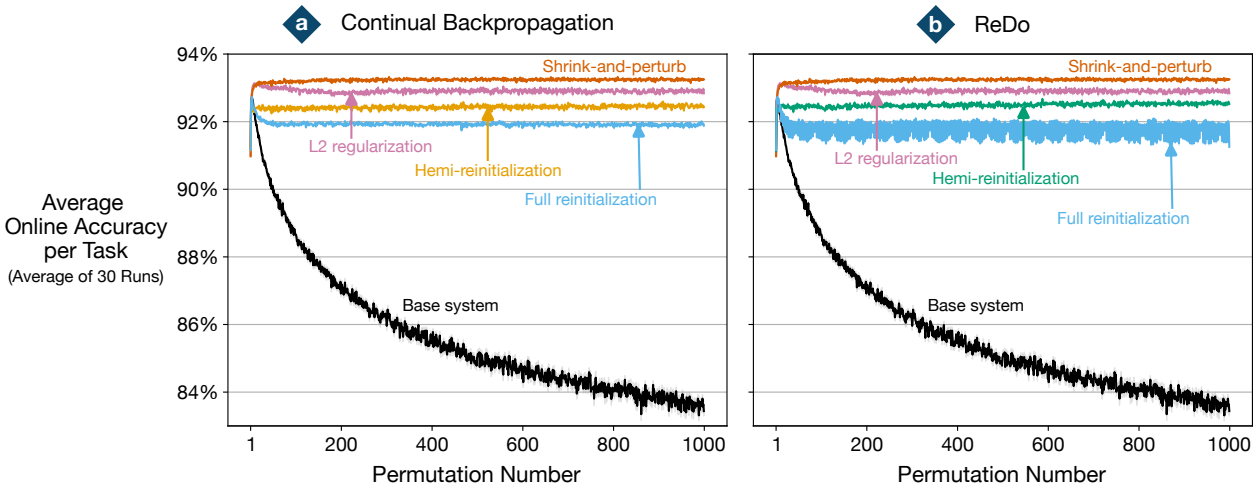

Figure 4.6: **a.** Performance of continual backprop with hemi- or full reinitialization. **b.** Performance of ReDo with hemi- or full reinitialization. Each line represents the average of 30 independent runs, and the shaded regions indicate one standard error away from the average. In either ReDo or continual backprop, full reinitialization resulted in lower performance than hemi-reinitialization. Hyperparameter values are provided in Table A.7.

Hence, full reinitialization may improve performance.

The goal of the following experiment is to study how the reinitialization method affects the performance of selective unit reinitialization algorithms in the Permuted MNIST problem. The base system is a ReLU network with three hidden layers and 100 units per layer trained using SGD. ReDo and continual backprop use either hemi- or full reinitialization with activation utility for ReDo and contribution utility for continual backprop, both of which performed equivalently in ReLU networks (see Figure 4.4). L2 regularization and shrink-and-perturb are included as baselines.

Full reinitialization resulted in lower performance in both ReDo and continual backprop. However, the learning curves show qualitatively different behaviours. In the case of continual backprop, full reinitialization resulted in lower performance, although performance remained largely stable. In ReDo, on the other hand, full reinitialization not only resulted in lower performance but also in sudden drops immediately after reinitializing units.

More insight into ReDo's behaviour can be obtained by zooming in on its performance. Figure 4.7 shows the performance of ReDo in the first 200 permutations. The erratic performance observed in ReDo with resample reinitialization is due to the reinitialization frequency. After tuning, the reintialization frequency that resulted in the best performance was $8,192$, or 2^{13}. That means a reinitialization step was executed every $8,192$ parameter updates. Since tasks consisted of only $2,000$ parameter updates, such a high reinitialization frequency meant that some tasks did not include a reinitialization step. The system executed a reinitialization step every 4.096 tasks,

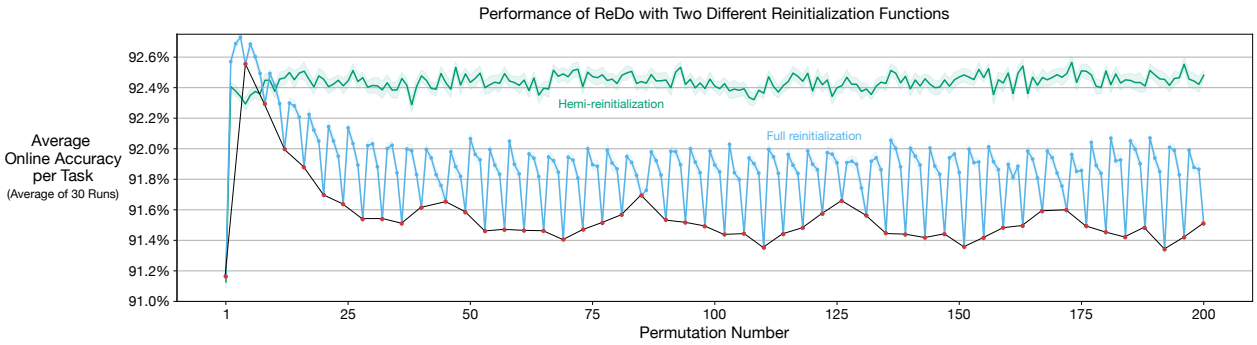

Figure 4.7: Performance of ReDo with hemi- or full reinitialization methods. Each line represents the average of 30 independent runs, and the shaded regions indicate one standard error away from the average. ReDo with full reinitialization showed drops in performance on tasks that included a reinitialization step, as indicated by the red dots in the plot.

resulting in a performance drop approximately every four tasks. Tasks where ReDo executed a reinitialization step are marked as red dots in Figure 4.7. Reinitialization steps also coincided with performance drops, highlighting the detrimental effect of full reinitialization. Additionally, looking only at the tasks where the learning system reinitialized units (black line in Figure 4.7), another pattern emerges. The performance in the tasks with reinitialization also shows peaks and valleys. These fluctuations correspond to when reinitialization was executed within the task. During the valleys, reinitialization was executed halfway through the task, whereas in the peaks, reinitialization was executed close to the start or the end of the task. Finally, ReDo with hemi-reinitialization did not exhibit sudden drops in performance and used a much lower reinitialization frequency—16 parameter updates between reinitialization steps.

As a final observation, full reinitialization yielded slightly higher performance on the first five tasks in both ReDo and continual backprop. The effect is minimal, but it still raises the question of whether it is possible to modulate the amount of noise to achieve higher performance that is also steady. Nevertheless, hemi-reinitialization seems the most sensible choice of reinitialization method so far.

4.4 Definition of a unit

This chapter presents results only for learning systems based on fully-connected networks. Ignoring bias terms for simplicity, a unit in a fully-connected layer is a node in the computational graph of the network corresponding to the activation: $\boldsymbol{h}_l[i] = g(\mathbf{W}_{l-1}[i,:] \cdot \mathbf{x})$, where l is the index of the layer, $\mathbf{W}_{l-1}[i,:]$ are the input weights of the unit, and $\mathbf{x}$ is the input to the layer. This unit is connected to the next layer through its output weights $\mathbf{W}_l[:,i]$. Thus, to reinitialize the unit, new

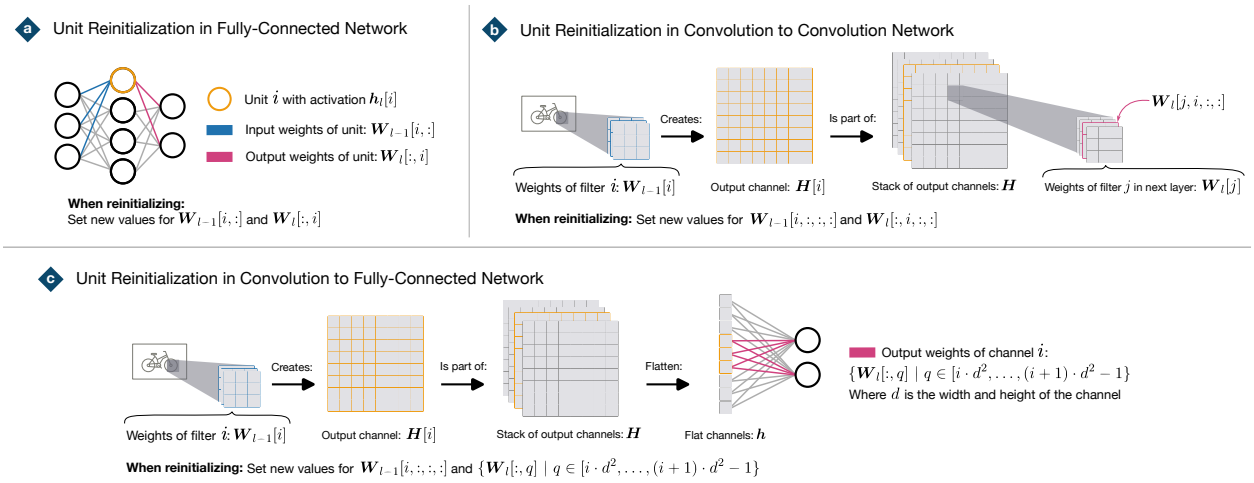

Figure 4.8: **a.** Illustration of unit reinitialization in a fully-connected network. **b.** Illustration of unit reinitialization in a convolutional layer followed by a convolutional layer. **c.** Illustration of unit reinitialization in a convolutional layer followed by a fully-connected layer.

values are assigned to the input weights, $\mathbf{W}_{l-1}[i,:]$, and the output weights, $\mathbf{W}_l[:,i]$. This process is illustrated in Figure 4.8a.

The notion of a unit varies across different types of architectures. For example, in a convolutional layer, output channels of convolution operations can be considered the basic units in the layer. Each output channel is an entire matrix of features created by sliding a single filter across a given image, $\mathbf{X} \in \mathbb{R}^{C_{in} \times d \times d}$, where C_{in} is the number of input channels and d is both the height and width of the image. A filter i has a corresponding weight tensor, $\mathbf{W}_{l-1}[i] \in \mathbb{R}^{C_{in} \times k \times k}$, where k is both the height and width of the filter. Assuming the convolutional operation leaves the dimensions of the image unchanged, the output of the i-th filter is a channel, $\boldsymbol{H}[i] \in \mathbb{R}^{d \times d}$. Channels are stacked into a tensor, $\boldsymbol{H} \in \mathbb{R}^{C_{out} \times d \times d}$, where C_{out} is the number of filters in the layer. The output channels interact with the next layer differently depending on the type of layer.

When the next layer is another convolutional layer, each output channel is multiplied by a matrix in each filter in the next convolutional layer. For example, for filter j in the next layer with weights $\mathbf{W}_l[j] \in \mathbb{R}^{C_{out} \times k \times k}$, $\mathbf{W}_l[j,i] \in \mathbb{R}^{k \times k}$ multiplies the channel corresponding to the i-th filter of the preceding layer. Thus, reinitializing unit i involves setting new values for $\mathbf{W}_{l-1}[i,:,:,:]$ and $\mathbf{W}_l[j,i,:,:]$ for each filter j in the next layer, i.e., $\mathbf{W}_l[:,i,:,:]$. This process is illustrated in Figure 4.8b.

When a convolutional layer is connected to a fully-connected layer, then the output channels are first flattened into a vector $\boldsymbol{h} \in \mathbb{R}^{C_{out} \cdot d^2}$, where the first d^2 entries correspond to the first filter, the following d^2 entries to the second filter, and so on. The flattened vector is connected to

the fully-connected layer through a weight matrix $\mathbf{W}_l \in \mathbb{R}^{d_{fc} \times C_{out} \cdot d^2}$, where d_{fc} is the number of units in the fully-connected layer. Reinitializing unit i involves setting new values for $\mathbf{W}_{l-1}[i, :, :, :]$ and for each entry in $\{\mathbf{W}_l[:, q] \mid q \in [i \cdot d^2, \ldots, (i+1) \cdot d^2 - 1\}$. See Figure 4.8c for a graphical representation.

The convolutional layer example illustrates that, even though it is simple to reason in terms of units in an abstract sense, the concrete implementation varies substantially depending on the network's structure.

Furthermore, introducing additional operations may warrant reconsidering what constitutes a unit in the network. For example, consider a fully-connected neural network that uses layer normalization after the activations of each hidden layer. In such a case, the activation of layer l is $\boldsymbol{h}_l = g(\mathbf{W}_{l-1}\boldsymbol{h}_{l-1} + \mathbf{b}_{l-1})$ and the normalized activation is $\overline{\boldsymbol{h}}_l = \frac{\boldsymbol{h}_l - \overline{h_l}}{\mathrm{s}_{\boldsymbol{h}_l} + \epsilon} \odot \boldsymbol{\gamma} + \boldsymbol{\beta}$, where $\overline{h_l}$ is the sample average of the activations, $\mathrm{s}_{\boldsymbol{h}_l}$ the sample standard deviation, ϵ a small number to prevent division by zero, and $\boldsymbol{\gamma}$ and $\boldsymbol{\beta}$ are learnable parameters. In this case, one could define a unit in two different ways; the unit could correspond to the operation $\boldsymbol{h}_l$ or the operation $\overline{\boldsymbol{h}}_l$. The next experiment illustrates how the definition of a unit may affect performance by using $\boldsymbol{h}_l$ as the corresponding operation of the unit—the utility is computed *based on activations*—or $\overline{\boldsymbol{h}}_l$ as the corresponding operation of the unit—the utility is computed *based on normalized activations*. The base system is a fully-connected network with three hidden layers and ReLU activations, with layer norm used after each ReLU activation. In this experiment, continual backprop uses contribution utility and ReDo activation utility, and both utilize hemi-reinitialization. When reinitializing a unit, the corresponding entries in $\boldsymbol{\gamma}$ and $\boldsymbol{\beta}$ are reinitialized to one and zero, respectively.

When the utility is computed based on activations (Figure 4.9a), ReDo had steady performance, whereas continual backprop had a slight decrease in performance over time. On the other hand, when the utility is computed based on normalized activations (Figure 4.9b), both ReDo and continual backprop had a steady decrease in performance close to the performance of the base system. In other words, ReDo and continual backprop were significantly less effective at maintaining plasticity when using normalized activations for computing utilities. Thus, even when the architectures are identical, the choice of what constitutes a unit may significantly affect the effectiveness of selective unit reinitialization algorithms in maintaining plasticity.

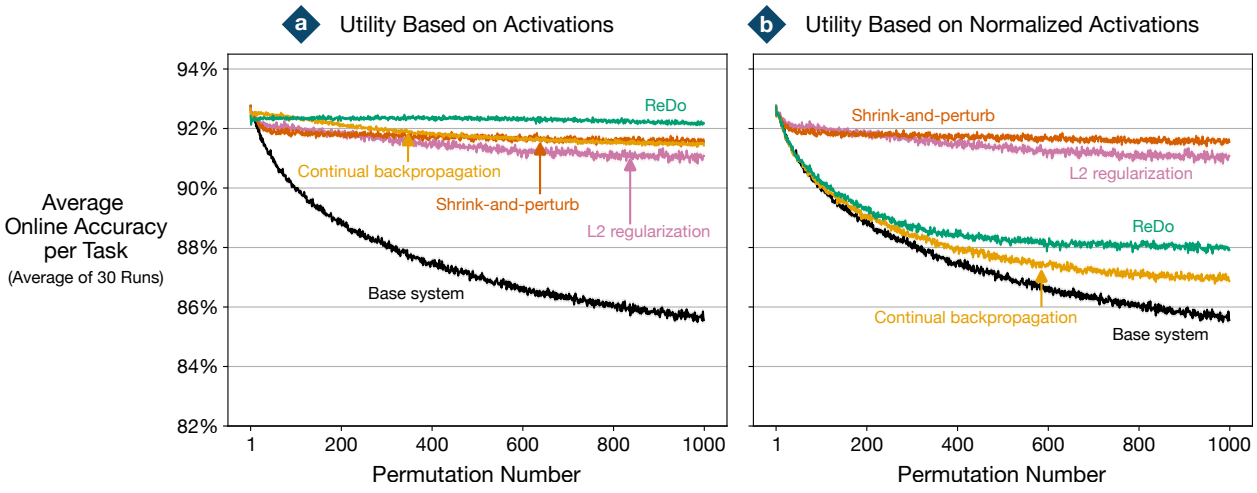

Figure 4.9: **a.** Performance of continual backprop and ReDo when performing reinitialization after the activation. **b.** Performance of continual backprop and ReDo when performing reinitialization after normalization. Each line represents the average of 30 independent runs, and the shaded regions indicate one standard error away from the average. In both continual backprop and ReDo computing utilities based on normalized activations resulted in a steady decrease in performance. Hyperparameter values are provided in Table A.8.

4.5 Related Literature on Reinitialization Algorithms and Future Work

This section discusses prior literature on reinitialization algorithms, pruning criteria, utility measures, and reinitialization methods. Many of the ideas discussed here can be used in the selective unit reinitialization algorithm (Algorithm 1). Thus, the topics discussed in this section not only provide context for past literature but also provide avenues for future work.

Reinitialization algorithms

The idea of searching over the space of units in a network to produce learning systems that can produce increasingly accurate solutions has a long history. The first instantiation of this idea dates back almost 70 years with Selfridge's Pandemonium architecture (Selfridge, 1958). Selfridge's architecture is strikingly similar to continual backprop but without gradient descent. The architecture involved a network of units that generated predictions to match target outputs and maximize a score function, and pruned the least useful units to make space for new units. Klopf and Gose (1969), Holland and Reitman (1977), and Kaelbling (1993) all describe architectures similar to Pandemonium that do not employ learning by gradient descent but instead search over the space of units in the network by pruning and generating new units. Mahmood and Sutton (2013) later

proposed a system that combined search over the space of units with gradient descent in shallow networks with a linear threshold unit. Both Dohare (2020) and Rahman (2021) proposed extensions of Mahmood and Sutton's work for deep neural networks with ReLU activations. These two extensions were also the first examples of selective reinitialization systems used to enable learning from a non-stationary stream of data without loss of plasticity. Dohare et al. (2021) proposed the first iteration of continual backpropagation, and Dohare et al. (2024) proposed an updated version using contribution utility. ReDo was proposed by Sokar et al. (2023) as an alternative to continual backpropagation to prevent the accumulation of frozen units. Self-normalized resets is the latest reinitialization algorithm in the plasticity literature proposed by Farias and Jozefiak (2025). Self-normalized resets reinitializes units with low activation by detecting changes in the pattern of activation of each unit.

A parallel line of work to algorithms that reinitialize units in a network is the idea of generating new units and constructing an increasingly complex network over time. The most notable example of this type of learning system is the cascade correlation proposed by Fahlman et al. (1988). The cascade correlation starts with no units and slowly builds deeper networks by training candidate units via gradient descent and adding the most promising ones to the network. Once a unit is added to the network, its input connections are frozen, and it serves as input to new candidate units to build increasingly complex representations. The cascade correlation performs a similar search to selective unit reinitialization, except for a few key differences. First, units are never removed from the network to make space for new units. Second, new units are not added to the network immediately. Instead, a pool of candidate units is trained without affecting the output of the network; then, the most promising candidate is added to the network. Lastly, units added to the main network have their input connections frozen, preventing any changes in the pattern of their activation as long as the data distribution remains stationary. The idea of constructing increasingly complex networks has been extended to convolutional networks and has also been used for preventing catastrophic forgetting in continual supervised learning (Marquez et al., 2018) and in reinforcement learning (Rusu et al., 2016). Since constructive learning systems involve continuous search over the parameter space, they are likely to maintain plasticity in continual learning. Thus, constructive learning systems may be capable of continual learning without forgetting and without loss of plasticity. However, their growing computational demands may be a limiting factor.

Reinitialization algorithms have also been proposed in response to the Lottery Ticket Hypothesis, which states that a randomly initialized dense network contains a sub-network capable of achieving comparable performance to the dense network when trained in isolation (Frankle and Carbin, 2019). The Lottery Ticket Hypothesis has led to the proposal of dynamic sparse training algorithms, which search over the parameter space of sparse networks to find a sub-network with good performance (Mocanu et al., 2018; Bellec et al., 2018; Evci et al., 2020). Dynamic sparse training algorithms prune and revive connections in sparse networks to find a good solution, and they

have been shown to make learning systems robust to input permutations in reinforcement learning (Grooten et al., 2023). Other algorithms inspired by the Lottery Ticket Hypothesis implement the search process by reinitializing layers, units, or weights in dense networks (Alabdulmohsin et al., 2021; Taha et al., 2021; Zhou et al., 2022). These reinitialization algorithms have been shown to be useful for improving fine-tuning performance (Li et al., 2020), making learning systems robust to label noise (Zaidi et al., 2023), and mitigating plasticity loss in reinforcement learning (Nikishin et al., 2022).

Pruning criteria

Pruning criteria have been used for two distinct applications: (1) for model compression and (2) for unit or weight reinitialization to create a search process over the parameter space complementary to gradient descent. For brevity, let us call the second pruning application *pruning for reinitialization*. The key difference between pruning for model compression and pruning for reinitialization is that the former prunes parameters after the network has learned, whereas the latter prunes them online as the network is learning. Dynamic sparse learning algorithms and selective reinitialization algorithms use pruning for reinitialization. Because the goal of pruning in both of these families of algorithms is the same, advances in pruning criteria in one family of algorithms can also benefit the other.

The latest pruning criterion proposed for reinitializing units is self-normalized resets introduced by Farias and Jozefiak (2025). Self-normalized resets can also be viewed as another instance of the general selective unit reinitialization algorithm in Algorithm 1. The idea of the algorithm is simple: count the number of time steps between activations of each ReLU unit—called *inter-firing time*—and reinitialize units whose inter-firing time is too large. A simplified version of self-normalized resets, called *fixed reset threshold*, keeps track of the inter-firing time of each unit and reinitializes the unit if its inter-firing time is greater than or equal to a *rejection threshold*, τ, which is treated as a hyperparameter. The inter-firing time at step t is stored in a vector, $\boldsymbol{a}_{t,l}$, for each layer l. Algorithm 4 shows pseudo-code of fixed reset threshold, where the blue lines correspond to the pruning criterion of the algorithm. The fixed reset threshold algorithm has not been published and was shared with me through personal communication with Farias and Jozefiak. Nevertheless, it more succinctly illustrates the ideas of self-normalized resets. Self-normalized resets works in a similar fashion, but the inter-firing time is modelled as a Geometric distribution, and units are reinitialized if the observed inter-firing time is deemed too improbable.

Self-normalized resets and fixed reset threshold assume the network uses ReLU activations and exploits the fact that inactive ReLU units output zero most of the time. The design of the algorithms couples together the utility measure, pruning criterion, and activation function in the network. The pruning criterion checks whether the activation of ReLU units is zero and counts how

Algorithm 4 Fixed reset threshold with generic reinitialization method

Input: a fully-connected network with L layers parameterized by $\boldsymbol{\theta}_0 = \{\mathbf{W}_{0,0}, \mathbf{W}_{0,1}, \ldots, \mathbf{W}_{0,L}\}$
Input: a reinitialization method
Input: rejection threshold, τ
Initialize: $\boldsymbol{a}_{0,1}, \ldots, \boldsymbol{a}_{0,L}$ to zero

for each time step, $t \in \{1, 2, \ldots\}$ **do**
  Receive an observation-target pair, compute predictions, and observe losses
  Compute gradients of the losses with respect to $\boldsymbol{\theta}_{t-1}$ and update using SGD
  for each layer in the network, $l \in \{1, \ldots, L\}$ **do**
    Compute activations of the units in the layer: $\boldsymbol{h}_{t,l}$
    Set $\boldsymbol{I}$ to be an empty list
    for Unit i layer l **do**
      if $\boldsymbol{h}_{t,l}[i] \leq 0$ **then**
        Increase inter-firing time: $\boldsymbol{a}_{t,l}[i] = \boldsymbol{a}_{t-1,l}[i] + 1$
      else
        Reinitialized inter-firing time: $\boldsymbol{a}_{t,l}[i] = 0$
      if $\boldsymbol{a}_{t,l}[i] \geq \tau$ **then**
        Add i to set of indices of units to be reinitialized, $\boldsymbol{I}$
        Reinitialize inter-firing time: $\boldsymbol{a}_{t,l}[i] = 0$
    for $i \in \boldsymbol{I}$ **do**
      Set input, $\mathbf{W}_{t,l-1}[i, :]$ and output, $\mathbf{W}_{t,l}[:, i]$, weights using reinitialization method

many times such an event occurs. To decouple these three elements of the algorithm, it could check for a different event based on an arbitrary utility measure. For example, for any arbitrary utility vector, $\boldsymbol{u}_{t,l}$, the algorithm could check whether the utility of unit i is below a given threshold, ρ, i.e., $\boldsymbol{u}_{t,l}[i] \leq \rho$. Such a modification would add another hyperparameter to the algorithm, but it would also enable its use with arbitrary utility measures and activation functions. An interesting line of future work is to study how the performance of self-normalized resets varies with different choices of utility measures.

Utility measures

Measuring the utility of units and weights in neural networks has been studied across many areas of deep learning, leading to a wide variety of utility measures. These utility measures could be adapted for use in selective reinitialization algorithms and may improve performance. Utility measures have been based on information about the weights, activations, first-order derivatives with respect to the weights, and second-order derivatives with respect to the weights. There are a multitude of ways to use this information to measure the utility of units or connections.

Using the magnitude of the weights has been shown to be a competitive strategy to prune

parameters in a neural network (Han et al., 2015; Blalock et al., 2020). Further improvements can be achieved by considering the interconnections between weights across layers to account for how removing a single weight affects the preceding and following layers connected to it. The resulting utility measure aggregates the magnitude of all the interconnected weights (Park et al., 2020). This idea can be extended to the unit level by building a graph of interconnected units and aggregating the magnitudes of all units' weights (Fang et al., 2023). The first iteration of continual backprop also accounted for the interconnections among different structures in a network by including a unit's input and output weights in its utility measure (Dohare et al., 2021).

Unit activations have also been considered to measure the utility of units in neural networks. Molchanov et al. (2017) evaluated different utility measures when pruning convolutional neural networks and found that, at low compression rates, the average activation of a filter resulted in similar performance as more complex utility measures. In selective unit reinitialization algorithms, activation utility is among the most popular utility measures for choosing which units to reinitialize (Sokar et al., 2023; Farias and Jozefiak, 2025). The latest iteration of continual backprop utilizes the magnitude of the output weights, along with the unit's average activation, as a utility measure (Dohare et al., 2024).

Information about the first-order derivatives with respect to the weights is also a popular choice for measuring the utilities of units in a neural network (Blalock et al., 2020). In neural network pruning, the motivation for this utility measure is to approximate the difference in loss when excluding a unit from the network (Mozer and Smolensky, 1988; Molchanov et al., 2019), which follows similar derivations as the first-order utility in (4.6). This utility measure has also been used to prune weights before training begins, resulting in sparse networks with competitive performance to their dense counterparts (Lee et al., 2019). One recent innovation in selective reinitialization algorithms is to use the derivative of the loss with respect to the weights as a measure of utility. This utility measure was proposed by Liu et al. (2025) and is similar to the first-order utility used in this chapter, but without multiplying by the sum of the output weights. The resulting learning system was more effective at maintaining plasticity in networks that included layer normalization.

Using the first-order information as a measure of utility has also been a popular approach for preventing catastrophic forgetting. Elastic weight consolidation (Kirkpatrick et al., 2017), synaptic intelligence (Zenke et al., 2017), and memory aware synapses (Aljundi et al., 2018) are all continual learning algorithms that use first-order information to prevent forgetting. After learning a task, these methods measure the utility of the weights and add a regularization term to the loss that penalizes solutions that deviate too much from the values of weights with high utility. An interesting deviation from other first-order utility measures is introduced by memory aware synapses from Aljundi et al. (2018). Memory aware synapses uses first-order information to determine how the output of the network changes due to small parameter perturbations, rather than how the loss

changes, as other utility measures based on first-order information do. It is possible to follow this same line of thought and derive a utility measure analogous to the first-order utility presented in this chapter. Assuming the network has a single output unit, the corresponding utility measure is the first-order Taylor approximation of the difference in the network output when omitting unit i in layer l:

$$\begin{aligned} f_{\boldsymbol{\theta}}(\mathbf{x} \mid \boldsymbol{h}_l[i] = 0) - f_{\boldsymbol{\theta}}(\mathbf{x}) &\approx f_{\boldsymbol{\theta}}(\mathbf{x}) - \Big(\frac{\partial f_{\boldsymbol{\theta}}(\mathbf{x})}{\partial \boldsymbol{h}_l[i]}\Big)\boldsymbol{h}_l[i] - f_{\boldsymbol{\theta}}(\mathbf{x}) \\ &= -\sum_{j=1}^{d_l} \mathbf{W}_l[j,i] \frac{\partial f_{\boldsymbol{\theta}}(\mathbf{x})}{\partial \mathbf{W}_l[j,i]}, \end{aligned}$$

where d_l is the number of units in layer l. The utility measure estimated from samples is given by:

$$\boldsymbol{u}_l[i] = \left| \sum_{j=1}^{d_l} \mathbf{W}_l[j,i] \; \frac{1}{m} \sum_{t=1}^{m} \frac{\partial f_{\boldsymbol{\theta}}(\mathbf{x}_t)}{\partial \mathbf{W}_l[j,i]} \right|. \tag{4.7}$$

The merits of this utility measure have yet to be investigated.

Many utility measures go beyond first-order information and utilize information about the curvature of the loss to assign utility to connections or units in the network. The motivation behind these utility measures is to approximate the difference in loss when omitting a unit or connection from the network using a second-order Taylor approximation (LeCun et al., 1989; Hassibi et al., 1993; Dong et al., 2017). While the approximation of the difference in loss is more accurate, the computation of the Hessian matrix for these utility measures often requires further approximations or simplifying assumptions. Nevertheless, efficient approximations are often competitive with other approaches in neural network pruning. In continual learning, utility measures based on second-order information have been used to add noise to the network's parameters and scale down the step-size of important parameters to address plasticity loss and catastrophic forgetting together (Elsayed and Mahmood, 2024). In that same work, a utility measure based on the first-order approximation of the loss difference yielded similar performance to the second-order approximation.

Reinitialization methods

Reinitialization methods have not been widely studied in the literature. The few ideas proposed in the past involve mutating existing units to create new units (Holland, 1992), training candidate units separately from the main network (Fahlman and Lebiere, 1989), and imprinting on input patterns so new units have high activations for such inputs (Javed, 2025). Another promising approach is to design distributions with desirable characteristics for continual learning. While initialization distributions have proven useful for reinitialization, they were originally designed

for training networks from scratch. Perhaps further improvements can be achieved by designing distributions for the explicit purpose of reinitialization.

4.6 Discussion and Conclusion

This chapter introduced the idea of selectively reinitializing parts of the network to maintain plasticity. To implement selective reinitialization, the chapter presented a general algorithm for reinitializing units in fully-connected networks. This general unit reinitialization algorithm subsumes previously proposed algorithms such as ReDo and continual backprop. The general view of selective unit reinitialization algorithms enabled the study of different settings for utility measures, pruning criteria, and reinitialization methods.

In the first set of experiments in this chapter, we explored the effectiveness of continual backprop and ReDo at maintaining plasticity in a variety of deep learning systems in the Permuted MNIST problem. Through these studies, we found that the effectiveness of ReDo and continual backprop increased with the number of units in each layer of the network. In the same studies, we found that continual backprop and ReDo were effective at maintaining plasticity when using different optimizers from stochastic gradient descent, such as stochastic gradient descent with momentum and Adam. Lastly, we found that although continual backprop and ReDo can maintain plasticity when the network uses Leaky ReLU activations, they fail to do so with Tanh activations.

After the initial evaluation of ReDo and continual backprop, we studied how these algorithms behaved under different utility measures and reinitialization methods. The chapter investigated the effect of five different utility measures: activation utility, based on the activations of units; weight magnitude utility, based on the magnitude of the outgoing weights of units; contribution utility, based on the activation and magnitude of outgoing weights; first-order utility, based on the first-order derivative of the loss with respect to the unit's activation; and random utility, which were random values sampled uniformly from [0,1]. In networks with ReLU activations, contribution, activation, and first-order utilities performed similarly, but weight magnitude utility performed poorly, failing to maintain plasticity in ReDo and plateauing at a lower level in continual backprop. Surprisingly, random utility performed only slightly lower than contribution, activation, and first-order utility, suggesting that even random reinitialization can maintain plasticity. Differences among activation, first-order, and contribution utilities became more pronounced across networks with different activation functions. In Leaky ReLU networks, first-order utility resulted in stable but slightly lower performance than activation utility in ReDo and contribution utility in continual backprop. In Tanh networks, first-order utility resulted in stable performance in both ReDo and continual backprop, suggesting that the previous failure to maintain plasticity in such networks was due to the choice of utility measure.

As a final evaluation, we studied different reinitialization methods in continual backprop and ReDo. We compared hemi-reinitialization, which samples new values from the initialization distribution for input weights and sets output weights to zero, against full reinitialization, which samples new values for input and output weights. In continual backprop, full reinitialization resulted in lower performance than hemi-reinitialization, yet performance remained stable. In ReDo, on the other hand, full reinitialization resulted in unstable performance that was lower than that of hemi-reinitialization. However, in both algorithms, full reinitialization achieved slightly higher performance than hemi-reinitialization on the first five tasks of the experiment. It may be possible to achieve higher performance by using random initialization on input and output weights while controlling the amount of noise introduced by the reinitialization.

This chapter did not discuss which setting of selective unit reinitialization was more effective at maintaining plasticity. Specifically, we did not address whether ReDo is more effective than continual backprop, or vice versa. The question of which algorithm is better was less important than studying how different settings affected the performance of selective unit reinitialization algorithms. Moreover, the answer to the question of which algorithm is better is likely dependent on the setting of interest. For example, in ReLu and Leaky ReLU networks, ReDo and continual backprop had comparable performance. When using layer normalization after ReLU activations, ReDo had slightly higher performance than backprop (Figure 4.9). Still, when using Tanh activations, continual backprop performed higher than ReDo when using first-order utility (Figure 4.5b). Similarly, the chapter did not discuss whether shrink-and-perturb was more effective than selective reinitialization algorithms at maintaining plasticity. In this case, the comparison is not relevant since shrink-and-perturb and selective reinitialization can be used together in the same learning system. Nevertheless, shrink-and-perturb did not always perform better than selective reinitialization. For example, when using Adam optimizer, shrink-and-perturb performed lower than ReDo and continual backprop (Figure 4.2b). Similarly, when using Tanh activations, shrink-and-perturb performed lower than continual backprop with first-order utility (Figure 4.5b). Throughout this chapter, no algorithm dominated the others, so claims about state-of-the-art performance are less relevant than studying in which settings these algorithms are expected to work.

The evaluations in this chapter concluded with a discussion of how the definition of a unit affects the implementation of selective unit reinitialization and its performance. In ReLU networks with layer normalization after the activations, the performance of ReDo and continual backprop varied significantly based on whether the utility was computed based on the activations or the normalized activations (Figure 4.9). A similar question could arise in other architectures. For example, in convolutional networks, ReLU activations are often followed by a pooling operation that aggregates pixels within a channel. In such an architecture, when using selective unit reinitialization, the question is whether to compute the utility based on the output of the ReLU activation or the output of the pooling operation. Performing selective reinitialization at the level of units in the network

requires the consideration of all these nuances. The following chapter simplifies the application of selective reinitialization by implementing the idea at the level of the weights.

Chapter 5

Selectively Reinitializing Weights to Maintain Plasticity

The previous chapter established selective reinitialization as a viable approach for preventing plasticity loss. The results showed that algorithms that reinitialize units maintain plasticity across a wide range of deep learning systems. A crucial consideration for applying these algorithms is defining what constitutes a unit, which can yield remarkably different results.

This chapter extends the idea of selective reinitialization by working at the level of weights within a network. The corresponding algorithm, *selective weight reinitialization*, does not require reasoning about units in a network, offering increased flexibility. Weight reinitialization is a previously unexplored approach; therefore, part of this investigation is dedicated to uncovering settings of the algorithm that are effective in maintaining plasticity. I then examine the tradeoffs between reinitializing units against weights in a wide range of deep learning systems. Experiments are performed using systems based on fully-connected networks in the Permuted MNIST problem.

The results show that multiple configurations of selective weight reinitialization successfully preserve plasticity. Moreover, I uncover several settings where reinitializing weights is more effective than units at maintaining plasticity. However, a drawback of this approach is that it lacks mechanisms for preventing disruptions to the network's output, often resulting in less stable training compared to unit reinitialization. Overall, this chapter further solidifies selective reinitialization as a general approach for preventing plasticity loss in systems based on fully-connected networks.

This chapter builds on collaborative work published at Conference on Lifelong Learning Agents in 2025 (Hernandez-Garcia et al., 2025), presenting my contributions along with additional analyses that provide deeper insight into the algorithm's behavior.

5.1 Motivation

As presented in the previous chapter, reinitializing units in a network was a successful approach for maintaining plasticity in the Permuted MNIST continual learning problem. With an appropriate choice of a utility measure, systems that selectively reinitialize units were capable of maintaining plasticity in a wide variety of settings. However, as presented near the end of the previous chapter, two considerations make it difficult to implement the idea of selective unit reinitialization to any arbitrary architecture. First, the specific connectivity pattern of the network results in different notions of what it means to be a unit, each notion involving distinct ways to reinitialize the unit's weights. For example, a unit in a fully connected layer is different from a unit in a convolutional layer (see Figure 4.8 for an illustration). Moreover, reinitializing the outgoing weights of a unit in a convolutional layer is handled differently depending on the next layer (see Figures 4.8b and 4.8c). Second, operations related to the activation of the unit may affect the effectiveness of unit reinitialization algorithms. For example, in a fully-connected network with layer normalization, computing the utility based on the activations resulted in significantly different performance compared to computing the utility based on the normalized activations (see Figure 4.9).

The difficulties in applying selective unit reinitialization to any arbitrary neural network are only practical and could be overcome with enough engineering. Nevertheless, reinitialization at the level of weights in a network sidesteps these difficulties, making it easier to implement. When reinitializing weights, each weight is treated the same regardless of the network's connectivity pattern. Moreover, working at the level of the weights also offers increased granularity when selecting and pruning parts of the network. The increased granularity may allow the pruning of a smaller portion of the network, making reinitialization a less disruptive operation.

Reinitializing weights may also be a more biologically plausible mechanism than reinitializing units. Biological neurons have been observed to periodically prune a proportion of their synaptic connections, while simultaneously growing new connections at the same rate (Kasai et al., 2021). This process is analogous to the continuous initialization of weights in reinitialization algorithms. The synaptic pruning and growing process in biological neurons suggests that reinitialization may facilitate continual learning in networks of interconnected neurons. Notably, reinitialization occurs at the level of synaptic connections, equivalent to weights in artificial neural networks, rather than at the level of neurons. This fact should not be taken as a strong argument in favour of reinitializing weights, but solely as inspiration for developing new algorithms.

Algorithm 5 Selective weight reinitialization

Input: any neural network
Input: a pruning criterion
Input: a utility measure
Input: a reinitialization method
Input: reinitialization frequency τ and reinitialization factor ρ

for each time step, $t \in \{1, 2, \dots\}$ **do**
    Sample a mini-batch of data, compute predictions, and observe losses
    Compute gradients and update parameters using SGD
    if t is a multiple of τ **then**
        for each weight matrix and vector in the network **do**
            Measure the utility of weights
            Use pruning criterion with reinitialization factor, ρ, to select weights to reinitialize
            Set new values for the selected weights using reinitialization method

5.2 Selective Reinitialization of Weights

This section introduces selective weight reinitialization, a general algorithm for reinitializing weights in a neural network. Selective weight reinitialization supplements learning with backpropagation and stochastic gradient descent (SGD). Just as the selective unit reinitialization algorithm presented in the previous chapter, selective weight reinitialization consists of three main components: a utility measure, a pruning criterion, and a reinitialization method. The main difference is that each of these three components is applied at the level of the weights.

Selective weight reinitialization works as follows. Every certain number of parameter updates, the algorithm executes a reinitialization step in which the utility of the weights is measured, and the least useful weights are pruned and subsequently assigned new values. Specifically, every τ steps, a reinitialization step commences. The hyperparameter τ is called the *reinitialization frequency*. During a reinitialization step, the utility of each weight in a weight matrix or bias vector in the network is measured according to a specific utility measure. The utility measure assigns real values to weights to indicate their importance, with larger positive values indicating greater importance. After measuring the utility of weights, a pruning criterion compares the utilities of the weights within the same weight matrix or bias vector and returns a set of indices for the selected weights to be reinitialized. The pruning criterion introduces another hyperparameter, called the *reinitialization factor* and denoted ρ, that modulates the number of weights pruned on each reinitialization step. Finally, a reinitialization method assigns new values to the weights selected by the pruning criterion. Algorithm 5 gives the pseudo-code for selective weight reinitialization.

Algorithm 5 provides pseudo-code for selective weight reinitialization with a generic utility measure, pruning criterion and reinitialization method. The next step in this chapter is to explore

various options for the core elements of selective weight reinitialization. Let us commence with the utility measure.

Utility measures

The role of a utility measure is to assign a real value, also called *utility*, to each weight in a matrix or vector to indicate its importance. The larger the utility of a weight, the more important it is to the network. For simplicity, this explanation describes selective weight reinitialization for weight matrices. However, in the experiments that follow, selective weight reinitialization is applied to every matrix and vector of parameters in the network. Let us denote the utility of the weights in a matrix as $\mathbf{U}$, with dimensions equal to those of the corresponding matrix.

This chapter examines two distinct utility measures commonly used in the pruning literature (Blalock et al., 2020). The first utility measure is simply the magnitude of the weight. For a weight matrix, $\mathbf{W}$, the utility of weight in the i-th row and j-th column is:

$$\mathbf{U}[i, j] = |\mathbf{W}[i, j]| \,. \tag{5.1}$$

Let us call this utility measure *weight magnitude utility*.

Another way to measure the utility of a weight is to calculate the difference in loss when the weight is omitted from the network. Let $f_{\boldsymbol{\theta}}$ be a network parameterized by a set of weight matrices, $\boldsymbol{\theta} = \{\mathbf{W}_0, \ldots, \mathbf{W}_L\}$, where L corresponds to the number of layers in the network. Let $\hat{\boldsymbol{y}} = f_{\boldsymbol{\theta}}(\mathbf{x})$ be the prediction of the network based on the observation, $\mathbf{x}$, and $\boldsymbol{y}$ be the target outputs. Lastly, let $\ell(\hat{\boldsymbol{y}}, \boldsymbol{y})$ be the loss between prediction and target and $\ell(\hat{\boldsymbol{y}}, \boldsymbol{y} \mid \mathbf{W}_l[i, j] = 0)$ be the loss when setting weight $\mathbf{W}_l[i, j]$ to zero. The difference,

$$\ell(\hat{\boldsymbol{y}}, \boldsymbol{y} \mid \mathbf{W}_l[i, j] = 0) - \ell(\hat{\boldsymbol{y}}, \boldsymbol{y}),$$

measures the impact that weight $\mathbf{W}_l[i, j]$ has on the loss while keeping all the other weights constant. Computing the difference in the equation above for each weight in the network is computationally expensive. To reduce computation, the difference can be approximated using a first-order Taylor approximation at the point $\mathbf{W}_l[i, j] = 0$:

$$\begin{aligned}
\ell(\hat{\boldsymbol{y}}, \boldsymbol{y} \mid \mathbf{W}_l[i, j] = 0) - \ell(\hat{\boldsymbol{y}}, \boldsymbol{y}) &\approx \ell(\hat{\boldsymbol{y}}, \boldsymbol{y}) + \frac{d\ell(\hat{\boldsymbol{y}}, \boldsymbol{y})}{d\mathbf{W}_l[i, j]}(0 - \mathbf{W}_l[i, j]) - \ell(\hat{\boldsymbol{y}}, \boldsymbol{y}) \\
&= -\mathbf{W}_l[i, j] \cdot \frac{d\ell(\hat{\boldsymbol{y}}, \boldsymbol{y})}{d\mathbf{W}_l[i, j]},
\end{aligned}$$

where $\frac{d\ell(\hat{\boldsymbol{y}}, \boldsymbol{y})}{d\mathbf{W}_l[i,j]}$ is the derivative of the loss with respect to $\mathbf{W}_l[i, j]$. The equation above is the difference for only a single observation-target pair. The quantity of interest is the expected difference

with respect to the data-generating distribution, p. Moreover, only those weights that have no impact on the loss should be considered for pruning. Thus, the utility of a weight is defined as:

$$\mathbf{U}_l[i,j] = \left| \mathbf{W}_l[i,j] \cdot \mathbb{E}_p \left[\frac{d\ell(f_{\boldsymbol{\theta}}(\mathbf{x}), \boldsymbol{y})}{d\mathbf{W}_l[i,j]} \right] \right|,$$

which can be estimated using a mini-batch of data of size m:

$$\mathbf{U}_l[i,j] = \left| \mathbf{W}_l[i,j] \cdot \frac{1}{m} \sum_{k=1}^{m} \frac{d\ell(f_{\boldsymbol{\theta}}(\mathbf{x}_k), \boldsymbol{y}_k)}{d\mathbf{W}_l[i,j]} \right|. \tag{5.2}$$

Let us refer to this utility measure as *first-order utility*.

Weight magnitude utility and first-order utility can be used to inform the pruning criterion. The next question to answer is: how should the algorithm select weights for reinitialization?

Pruning criteria

The role of a pruning criterion is to determine which and how many weights to reinitialize in the network. Calling the criterion a pruning criterion is a misnomer, as it implies that the weights and the corresponding connections are permanently removed from the network. Instead, one should think of a pruning criterion as a selection criterion, which chooses weights to reinitialize. Nevertheless, I chose to use the term pruning to maintain a link with the neural network pruning literature, which shares several concepts with selective reinitialization algorithms.

This chapter examines two pruning criteria inspired by those used in continual backpropagation and ReDo. At a high level, the pruning criterion inspired by continual backpropagation removes a fixed proportion of the parameters from each weight matrix in the network. Hence, the criterion is called *proportional pruning*. The pruning criterion inspired by ReDo removes all weights that fall below a threshold based on the mean utility of the weights in the matrix. Thus, the criterion is called *threshold pruning*.

In detail, proportional pruning operates as follows. Given a reinitialization factor, $\rho \in [0, 1]$, and a matrix of utilities, $\mathbf{U}$, proportional pruning removes $|\mathbf{U}| \cdot \rho$ weights from the matrix, where $|\mathbf{U}|$ is the number of weights in the matrix. To handle non-integer values for the number of weights to prune, proportional pruning handles the fractional part of $|\mathbf{U}| \cdot \rho$ by sampling from a Bernoulli distribution. The number of weights pruned on each reinitialization step, denoted κ, is Integer Part($|\mathbf{U}| \cdot \rho$) + Bernoulli(Fractional Part($|\mathbf{U}| \cdot \rho$)). Thus, in expectation, proportional pruning selects $|\mathbf{U}| \cdot \rho$ weights per reinitialization step. The pruning criterion returns a list of indices, $\boldsymbol{I}$, with entries corresponding to the index of the weights with the κ lowest utilities.

A different approach to removing a fixed proportion of weights from the network on each

reinitialization step is to prune weights that fall below a certain desired threshold. Threshold pruning implements this idea by first computing the average utility of the weights in the layer, denoted $\overline{u}$, and removing all the weights whose utility is less than or equal to $\rho \cdot \overline{u}$, for a reinitialization factor $\rho \geq 0$. The pruning criterion returns the list of indices $\boldsymbol{I} = \{(i, j) \mid \mathbf{U}[i, j] \leq \rho \cdot \overline{u}\}$.

The difference between proportional and threshold pruning is in the number of weights they prune per reinitialization step. In expectation, proportional pruning prunes $|\mathbf{U}| \cdot \rho$ weights per reinitialization step. On the other hand, threshold pruning may prune a variable number of weights depending on the distribution of the utilities. If there is an extreme outlier that significantly increases the average utility of the weights in the matrix, then most of the weights will be reinitialized. On the other hand, if all the utilities are higher than $\rho \cdot \overline{u}$, then threshold pruning may not remove any weights, assuming ρ is less than one.

The final step to obtain a concrete implementation of selective weight reinitialization is to define a reinitialization method.

Reinitialization methods

The reinitialization method assigns new values to the weights selected by the pruning criterion. When reinitializing a unit, the reinitialization method assigns new values to the unit's incoming and outgoing weights. The network's connectivity pattern changes how reinitialization is implemented when reinitializing units. When reinitializing a weight, the reinitialization method only needs to assign a new scalar value to the weight, without considering the network's connectivity, which simplifies implementation. However, there is a downside to this simplicity.

Reinitializing a weight or unit may affect the output of the network. Let us call this effect *impingement* and say instead that a new weight or unit may impinge on the network's output. While the impingement caused by a new weight or unit may positively or negatively impact performance, it is not possible to know a priori what the effect will be. Thus, it is desirable to choose reinitialization methods that incur as little impingement as possible. When reinitializing units, it is possible to mitigate the amount of impingement caused by the new unit by setting its output weights to zero. Such an option is not available when reinitializing weights, so the question becomes how to reinitialize weights without causing too much impingement.

This chapter presents two straightforward approaches to reinitializing weights. The first approach is to sample a new value from the initialization distribution. For example, assume the weights of a matrix $\mathbf{W}$ were initialized according to a normal distribution with mean μ and variance σ^2. When reinitializing weights corresponding to $\mathbf{W}$, the reinitialization method would sample a new value from $\mathcal{N}(\mu, \sigma^2)$. For this reason, let us refer to this approach as *resample reinitialization*. The second approach for reinitializing weights is to reinitialize the weight to the mean of the

initialization distribution. Using the same example as before, when reinitializing weights in $\mathbf{W}$, their value would be set to μ. Let us refer to this approach as *mean reinitialization*. Note that the mean of initialization distributions is often zero, so mean reinitialization is equivalent to setting new weights to zero in most networks. Also note that for weights initialized to a fixed value, such as bias terms and weight vectors in layer normalization, both resample and mean reinitialization set the value of new weights to the initial fixed value.

There are two motivations for selective reinitialization algorithms. The first motivation is to restore the initial conditions of the weights that allowed the network to learn and that were lost through the process of learning. The second motivation is to supplement gradient descent optimization by including an additional search process over the parameter space. Moreover, when reinitializing weights, one must also consider how to mitigate impingement. Resample reinitialization aligns with the two motivations for selective reinitialization algorithms. The introduction of randomness enables a parallel search process over the parameter space and also restores some of the initial conditions of the weights. On the other hand, mean reinitialization foregoes the motivations and focuses on mitigating impingement. Since both utility measures are proportional to the magnitude of the weight, the pruning criterion is likely to select weights with a small magnitude. Thus, setting the value of weights with low utility to zero is likely to incur little impingement. The following section examines which of the two reinitialization methods mitigates impingement and whether such a mitigation leads to better learning performance.

5.3 Effectiveness of Reinitializing Weights for Maintaining Plasticity

This section examines the impact of the choice of utility measure, pruning criterion, and reinitialization method on the effectiveness of selective weight reinitialization. This study focuses on three questions:

1. Is there any setting of selective weight reinitialization that maintains plasticity?
2. How does each choice affect the performance of the learning system?
3. Which combination of utility measure, pruning criterion, and reinitialization method results in the best performance?

The problem used to study these questions is the Permuted MNIST continual learning problem. The base learning system is a ReLU network with three hidden layers and 100 units per layer trained using SGD. The network is trained for one pass through the permuted dataset per task with a mini-batch size of 30, resulting in 2,000 parameter updates per task. Shrink-and-perturb and

L2 regularization are included as baselines of algorithms that are highly effective at maintaining plasticity in this problem.

To provide fair comparisons, the hyperparameters of each algorithm were tuned to maximize the average online accuracy over the span of 1,000 tasks. The hyperparameter search followed the same process as previous chapters: a wide search using a single run followed by a narrow search using multiple runs. As in the previous chapters, the step-size of the base system is tuned, and every other algorithm that modifies the base system uses the same step-size. The hyperparameters of each modification are also tuned. For each different combination of utility measure, pruning criterion, and reinitialization method, the reinitialization frequency, τ, and the reinitialization factor, ρ, were tuned. Moreover, reinitialization was performed on every weight matrix and bias vector in the network. The L2 regularization baseline used a tuned regularization factor. Lastly, for the shrink-and-perturb baseline, the variance of the noise was tuned, and the regularization factor was set to the same factor as the L2 regularization baseline. For more details on hyperparameter tuning, see Appendix A.

Let us start by examining the effect of the utility measure in proportional and threshold pruning with resample reinitialization. In addition to the two utilities presented in the previous section, the results include a random utility baseline. The random utility baseline assigns values sampled uniformly from $[0, 1]$ to each of the weights in the matrix or vector. Figure 5.1 shows the performance of each different utility measure.

Four of the settings of selective weight reinitialization maintained plasticity. First-order utility maintained a stable level of performance throughout the entire experiment with both proportional and threshold pruning. Using weight magnitude utility with proportional pruning resulted in a decrease in performance as the number of tasks increased (Figure 5.1a), and resulted in no difference compared to the base system when using threshold pruning (5.1b). Surprisingly, both settings of selective weight reinitialization with random utility maintained a steady level of performance and almost matched the performance of first-order utility. Another striking aspect of these results is the undulating pattern in the performance of some settings of selective weight reinitialization. Before examining the different behaviours of selective weight reinitialization settings in more detail, let us first consider how it performs with a different reinitialization mehtod.

Selective weight reinitialization was less effective at maintaining plasticity when using mean reinitialization (Figure 5.2). The settings of selective weight reinitialization that used weight magnitude utility or first-order utility saw a steady decrease in performance when using either of the pruning criteria. Notably, random utility was not only capable of maintaining plasticity but also yielded higher performance than resample reinitialization. To quickly verify this, observe the gap in performance between random utility and L2 regularization in Figure 5.1 and Figure 5.2; the gap is smaller when using mean reinitialization.

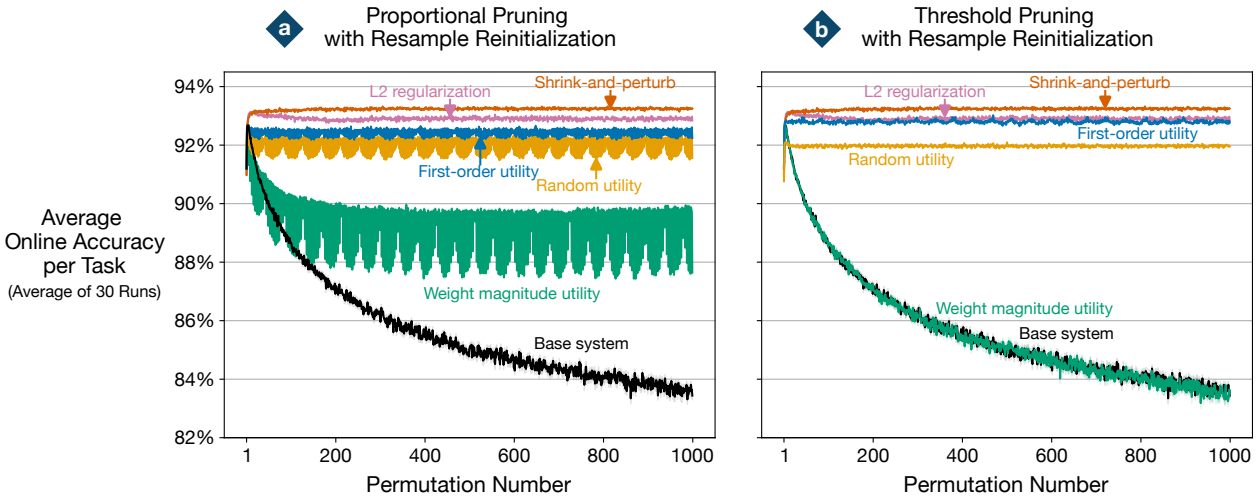

Figure 5.1: Performance of selective weight reinitialization with resample reinitialization and **(a)** proportional pruning or **(b)** threshold pruning with different utility measures. Each line represents the average of 30 independent runs, and the shaded regions indicate one standard error away from the average. Several settings of selective weight reinitialization maintained plasticity throughout the entire experiment. First-order utility had the highest average online accuracy of all utility measures. Hyperparameter values are provided in Table A.9.

The results presented so far provide answers to the three initial questions in this study. First, there were several settings of selective weight reinitialization capable of maintaining plasticity. Second, the choice of utility measure, pruning criterion, and reinitialization method had a significant impact on performance. None of the settings using the weight magnitude utility was capable of maintaining plasticity. First-order utility was only successful in maintaining plasticity when using resample reinitialization with threshold pruning, yielding higher performance than proportional pruning. Random utility proved surprisingly robust to the choice of pruning criterion and reinitialization method, maintaining plasticity in all four possible settings, although it resulted in higher performance when using mean reinitialization. The straightforward answer to the last question—what setting results in the best performance—is selective weight reinitialization with first-order utility, threshold pruning, and resample reinitialization. However, this answer would overlook other desirable characteristics in learning systems, such as robustness. Thus, let us delay answering this question until we gather more information about how selective weight reinitialization performs under different settings.

The initial results of the performance of different settings of selective weight reinitialization raise new questions. Why do some of the settings of selective weight reinitialization show undulating, cyclical patterns in performance? Why is random utility so robust to other algorithmic choices in selective weight reinitialization? Conversely, why are first-order utility and weight magnitude

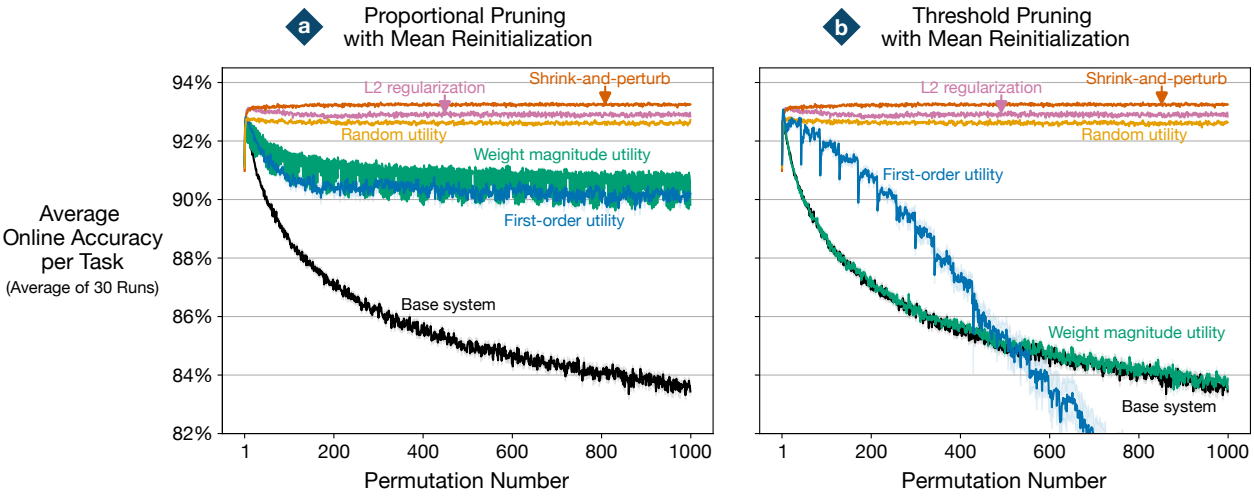

Figure 5.2: Performance of selective weight reinitialization with mean reinitialization and **(a)** proportional pruning or **(b)** threshold pruning with different utility measures. Each line represents the average of 30 independent runs, and the shaded regions indicate one standard error away from the average. Unlike with resample reinitialization, only settings using random utility maintained plasticity. Hyperparameter values are provided in Table A.9.

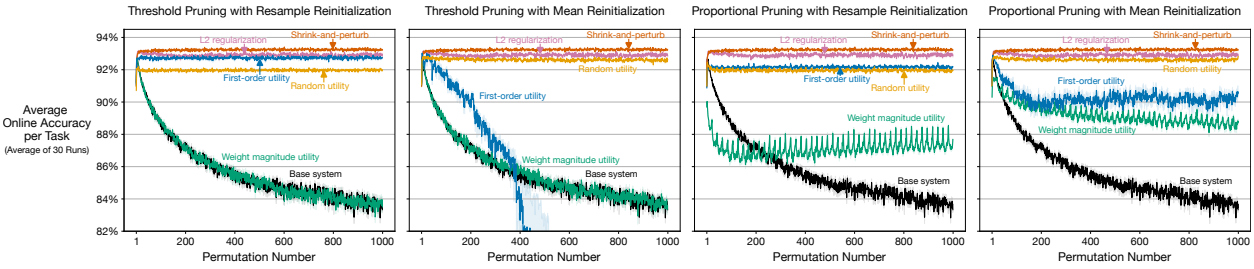

Figure 5.3: Performance of all settings of selective weight reinitialization with reinitialization frequency less than or equal to 1024. Each line represents the average of 10 independent runs, and the shaded regions indicate one standard error away from the average. Unlike with higher reinitialization frequencies, the undulating pattern in the performance of the learning systems is less pronounced or entirely removed. Hyperparameter values are provided in Table A.10.

utility so brittle to the choice of reinitialization method? The next part of the study focuses on answering these questions.

The undulating, cyclical pattern in the performance of some settings of selective weight reinitialization is reminiscent of the performance of ReDo with full reinitialization in the previous chapter (Figure 4.7). Just as with the performance of ReDo with full reinitialization, the undulating pattern in the performance is an artifact of the chosen reinitialization frequency. All settings with undulating performance used a reinitialization frequency higher than 2,000—the total number of

parameter updates per task. As a consequence, reinitialization did not occur on every single task. When reinitialization does occur, it happens at different points during tasks, sometimes early, at other times in the middle, and at times close to the end of the task, resulting in two cyclical patterns. First, a drop in performance is observed on those tasks that contain a reinitialization step. Second, the drop in performance is more pronounced when the reinitialization step is executed in the middle of the task. To further demonstrate how the undulating performance is an artifact of the reinitialization frequency, Figure 5.3 shows the performance of all the settings of selective weight reinitialization where the reinitialization frequency was capped to be at most 1,024 during the grid search. The undulating pattern in performance was reduced or removed. The undulating pattern persisted in some settings because even with a frequency of 1,024, reinitialization happens at different points within a task, and some tasks may include more than one reinitialization step.

The hyperparameter settings explain the undulating patterns found in performance, but a result still stands out as surprising: the robust performance of random utility. This result is surprising because the random utility provides no information about the usefulness of each weight, yet using random utility prevented plasticity loss with every pruning and reinitialization method.

One possible reason random utility is so robust to the other settings of selective weight reinitialization is that it has no relationship to the value of the weight. Weight magnitude utility and first-order utility include a factor equal to the absolute value of the weight; thus, weights with small magnitude are more likely to be reinitialized. The dependence on the weight magnitude can result in a cycle where new weights are assigned a small value and are unable to grow sufficiently to prevent them from being reinitialized again—let us call this phenomenon *persistent reinitialization*. If a setting of selective weight reinitialization suffers from persistent reinitialization, one would expect the same subset of weights to be selected for reinitialization in consecutive reinitialization steps. Both mean and resample reinitialization may suffer from persistent reinitialization if weights are set too small, but mean reinitialization could result in a larger effect since it sets new weights to zero.

To verify whether persistent reinitialization is occurring, let us define a new measure corresponding to the percentage of weights reinitialized in consecutive reinitialization steps. Let $\boldsymbol{I}$ and $\boldsymbol{I}'$ be the set of indices selected for reinitialization in the current and previous reinitialization steps, respectively. The quantity

$$\text{Reinitialization Overlap} \doteq 100 \cdot \frac{|I' \cap I|}{|I|},$$

measures the percent of weights reinitialized in consecutive reinitialization steps. The value of the reinitialization overlap ranges from 0 to 100, with values close to 100 implying that most of the weights reinitialized in the current reinitialization step were also reinitialized in the previous one. Table 5.1 shows summaries for each setting of selective weight reinitialization presented in Figures

Table 5.1: Reinitialization summaries for different settings of selective weight reinitialization. Each number is the average of 30 runs, while the value in parentheses corresponds to the standard error. Columns where the standard error is omitted had standard errors lower than 0.2. Rows where the average online accuracy is bold correspond to settings of selective weight reinitialization that maintained plasticity.

Utility Measure	**Pruning Criterion**	**Reinitialization Method**	**Percent Reinitialized per Task**	**Average Reinitialization Overlap**	**Average Online Accuracy**
First-Order	Threshold	Resample	48.58	56.93	**92.79** (0.002)
		Mean	94.04	98.39	85.39 (0.38)
	Proportional	Resample	19.52	55.22	**92.42** (0.003)
		Mean	78.12	92.67	90.42 (0.11)
Weight Magnitude	Threshold	Resample	0.0038	0.16	85.66 (0.01)
		Mean	22.52	94.58	85.78 (0.02)
	Proportional	Resample	19.52	97.06	89.46 (0.02)
		Mean	19.52	97.03	90.88 (0.008)
Random	Threshold	Resample	19.53	5.00	**91.97** (0.001)
		Mean	9.76	5.00	**92.62** (0.002)
	Proportional	Resample	9.76	19.96	**91.99** (0.001)
		Mean	9.77	5.00	**92.62** (0.002)

5.1 and 5.2. The table shows the percent of weights reinitialized per task (*percent reinitialized per task*), the average reinitialization overlap per reinitialization step (*average reinitialization overlap*), and the online accuracy averaged over all the tasks in the experiment (*average online accuracy*). Rows where the average online accuracy is bold correspond to settings of selective weight reinitialization that maintained plasticity.

The summaries in Table 5.1 provide further insight into the behaviour of selective weight reinitialization. Using random utility consistently resulted in a low reinitialization overlap, implying less persistent reinitialization. Conversely, using other utility measures resulted in high reinitialization overlap. A reinitialization overlap higher than 90% was associated with settings that failed to maintain plasticity. The exception was weight magnitude utility with threshold pruning and resample reinitialization, which lost plasticity because the amount of reinitialization was extremely low. Moreover, using random utility rendered the choice of the pruning criterion irrelevant, as performance varied only with the reinitialization method.

The summaries in Table 5.1 also provide more insight into the effectiveness of mean reinitialization. When using first-order and weight magnitude utility, mean reinitialization resulted in a reinitialization overlap higher than 90% and a failure to maintain plasticity. With random utility, on the other hand, mean reinitialization resulted in higher average online accuracy than resample reinitialization. The high reinitialization overlap could explain the poor performance with mean reinitialization with first-order and weight magnitude utilities.

The next part of this chapter focuses on comparing selectively reinitializing weights against units in several learning systems. Evaluating every setting of selective weight reinitialization is prohibitively expensive because of the extensive amount of hyperparameter tuning. Thus, the next section only presents results with the most promising settings. Specifically, the next section presents the results of four settings:

1. first-order utility with threshold pruning and resample reinitialization, labelled *SWR first-order, threshold, resample* in the next section,
2. first-order utility with proportional pruning and resample reinitialization, labelled *SWR first-order, proportional, resample*,
3. random utility with proportional pruning and resample reinitialization, labelled *SWR random, proportional, resample*,
4. and random utility with proportional pruning and mean reinitialization, labelled *SWR random, proportional, mean*.

These settings were selected because they all successfully maintained plasticity in the previous evaluation. Random utility with threshold pruning is omitted from the next evaluations since there was no difference in performance compared to proportional pruning.

5.4 Reinitializing Weights vs Units

This section compares two different approaches to implementing selective reinitialization in fully-connected networks: at the level of weights and at the level of units. The primary objective of this section is to identify settings where the two approaches are equally effective in maintaining plasticity and those where their effectiveness differs. This study is limited to fully-connected layers, but continues in the following chapters using other, more complex architectures.

In the previous section, we identified four different settings of selective weight reinitialization that were effective at maintaining plasticity. What setting should be used for comparing against unit reinitialization algorithms? The answer is simple: the one that performs the best with each base system. For each of the different base systems introduced next, we first compare the performance of the four settings of selective weight reinitialization. Then, the setting or settings that perform best are used to compare against ReDo and continual backpropagation. This approach for comparing reinitializing units against weights also provides more insight into the performance of selective weight reinitialization in different learning systems.

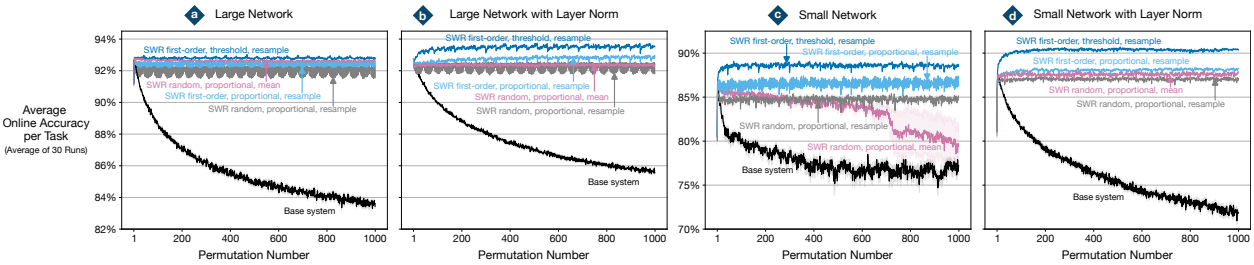

Figure 5.4: Performance of four settings of selective weight reinitialization in ReLU networks with three hidden layers and **(a)** 100 units per layer, **(b)** 100 units per layer and layer norm applied after activations, **(c)** 10 units per layer, and **(d)** 10 units per layer with layer norm applied after activation. Each line corresponds to the sample average of 30 independent runs. The shaded regions indicate one standard error away from the sample average. Selective weight reinitialization (SWR) with first-order utility, threshold pruning, and resample reinitialization consistently outperformed the other settings. Hyperparameter values are provided in Tables A.11 and A.12.

The problem used for comparing reinitializing units against weights is the Permuted MNIST problem. The validity of the comparisons presented in this section heavily relies on the hyper-parameter settings of each algorithm. Just like in previous evaluations, each algorithm is built upon a base system. All optimizer hyperparameters were tuned using grid search for the base system. Other algorithms used the same optimizer settings as the base system, with additional algorithm-specific hyperparameters also tuned via grid search.

We commence our studies by revisiting base learning systems where unit reinitialization algorithms were less effective at maintaining plasticity. Specifically, continual backprop and ReDo were less effective when the network had a small number of units and when the network included layer normalization. The first set of comparisons studies whether the effectiveness of selective weight reinitialization is also diminished in these settings. These evaluations include four base learning systems: a ReLU network with three hidden layers and 100 units per layer—*large network* system, the same architecture with only 10 hidden units per layer—*small network* system, the large network system with layer norm after activations—*large network with layer norm* system, and the small network system with layer norm after activations—*small network with layer norm* system.

We begin our analysis by examining the effectiveness of different settings for selective weight reinitialization when added to the four base systems. Most of the settings of selective weight reinitialization maintained plasticity in the four different systems. When extending the large network system or the systems with layer norm (Figures 5.4a, 5.4b, and 5.4d), all four settings of selective weight reinitialization preserved plasticity, showing stable performance rather than the continuous decline observed in the base systems. However, when extending the small network system (Figure 5.4c), selective weight reinitialization with random utility and mean reinitialization showed a steady

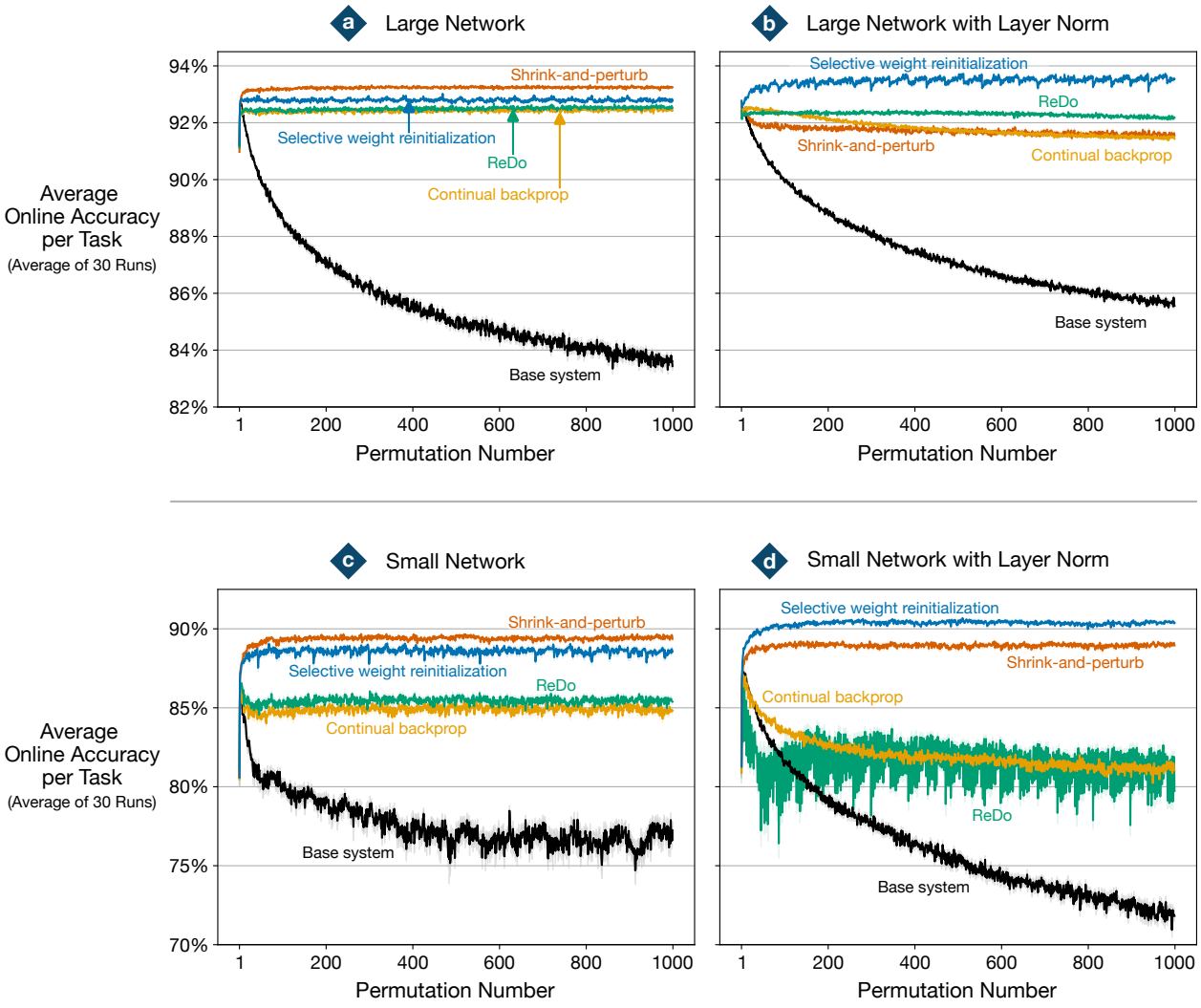

Figure 5.5: Performance of reinitialization algorithms in ReLU networks with three hidden layers and **(a)** 100 units per layer, **(b)** 100 units per layer and layer norm applied after activations, **(c)** 10 units per layer, and **(d)** 10 units per layer with layer norm applied after activation. Each line corresponds to the sample average of 30 independent runs. The shaded regions indicate one standard error away from the sample average. Selective weight reinitialization with first-order utility, threshold pruning, and resample reinitialization— abeled *selective weight reinitialization* in this plot—outperformed unit reinitialization approaches in all four architectures. Hyperparameter values are provided in Table A.13.

decrease in performance. Lastly, among all the settings, selective weight reinitialization with first-order utility, threshold pruning, and resample reinitialization consistently performed higher than other settings.

Having identified the best-performing setting for selective weight reinitialization, let us proceed

to compare it with unit reinitialization algorithms. Figure 5.5 shows the performance of ReDo and continual backprop with the four different base systems, as well as the performance of selective weight reinitialization with first-order utility, threshold pruning, and resample reinitialization. The figure also includes shrink-and-perturb as a baseline. For the unit reinitialization algorithms used with layer norm systems, the utility was computed based on the activations instead of the normalized activations, since such a setting yielded better results in the previous chapter (see Figure 4.9).

Unit reinitialization algorithms were similarly effective to selective weight reinitialization in the large network system (Figure 5.5a). In the large network with layer norm system (Figure 5.5b), continual backprop had a steady decrease in performance; ReDo also had a decrease in performance, but significantly less pronounced. Selective weight reinitialization exhibited an undulating performance pattern that remained constant throughout the entire experiment. In the small network system (Figure 5.5c), unit reinitialization had a steady performance after an initial decrease in performance. Once again, the performance of selective weight reinitialization stabilized at a higher level. Lastly, in the small network with layer norm system (Figure 5.5d), both unit reinitialization algorithms had a steady decrease in performance, whereas the performance of selective weight reinitialization was stable throughout the experiment. Selective weight reinitialization consistently maintained plasticity in all four base learning systems.

The results so far show that, in some learning systems, reinitializing weights is a more effective approach than reinitializing units. The next question of interest is whether there are learning systems where unit reinitialization is more effective than weight reinitialization. To study this question, let us revisit settings where unit reinitialization was shown to be effective in the previous chapter. We begin by verifying if selective weight reinitialization is robust to the choice of optimizer. The base systems in the following evaluations are ReLU networks with 100 units per layer trained using SGD with momentum or Adam. The shrink-and-perturb baseline uses SGDW and AdamW.

We commence by evaluating the performance of the four settings of selective weight reinitialization. Figure 5.6a shows the performance of selective weight reinitialization in the system trained with Adam and SGD with momentum. All four settings had stable performance in both optimizers. Random utility with resample reinitialization had the lowest performance among the four settings, whereas first-order utility with threshold pruning and resample reinitialization had the highest performance. The latter setting is used to compare against unit reinitialization algorithms in Figure 5.6b. In systems trained with Adam optimizer (left pane of Figure 5.6b), all three reinitialization algorithms performed equally well. In systems trained with SGD with momentum (right pane of Figure 5.6b), the performance of selective weight reinitialization was slightly higher than the performance of continual backprop and ReDo.

The previous chapter demonstrated that, with an appropriate choice of utility measure, unit

reinitialization algorithms can maintain plasticity when the network employs activation functions other than ReLU. The next evaluations verify if selective weight reinitialization is also capable of maintaining plasticity with different activation functions. The base systems in these evaluations are a network with Leaky ReLU activations and a network with Tanh activations, both of which contain three hidden layers with 100 units each. For the Tanh system, ReDo and continual backprop use first-order utility, whereas for the Leaky ReLU system, they use activation and contribution utility,

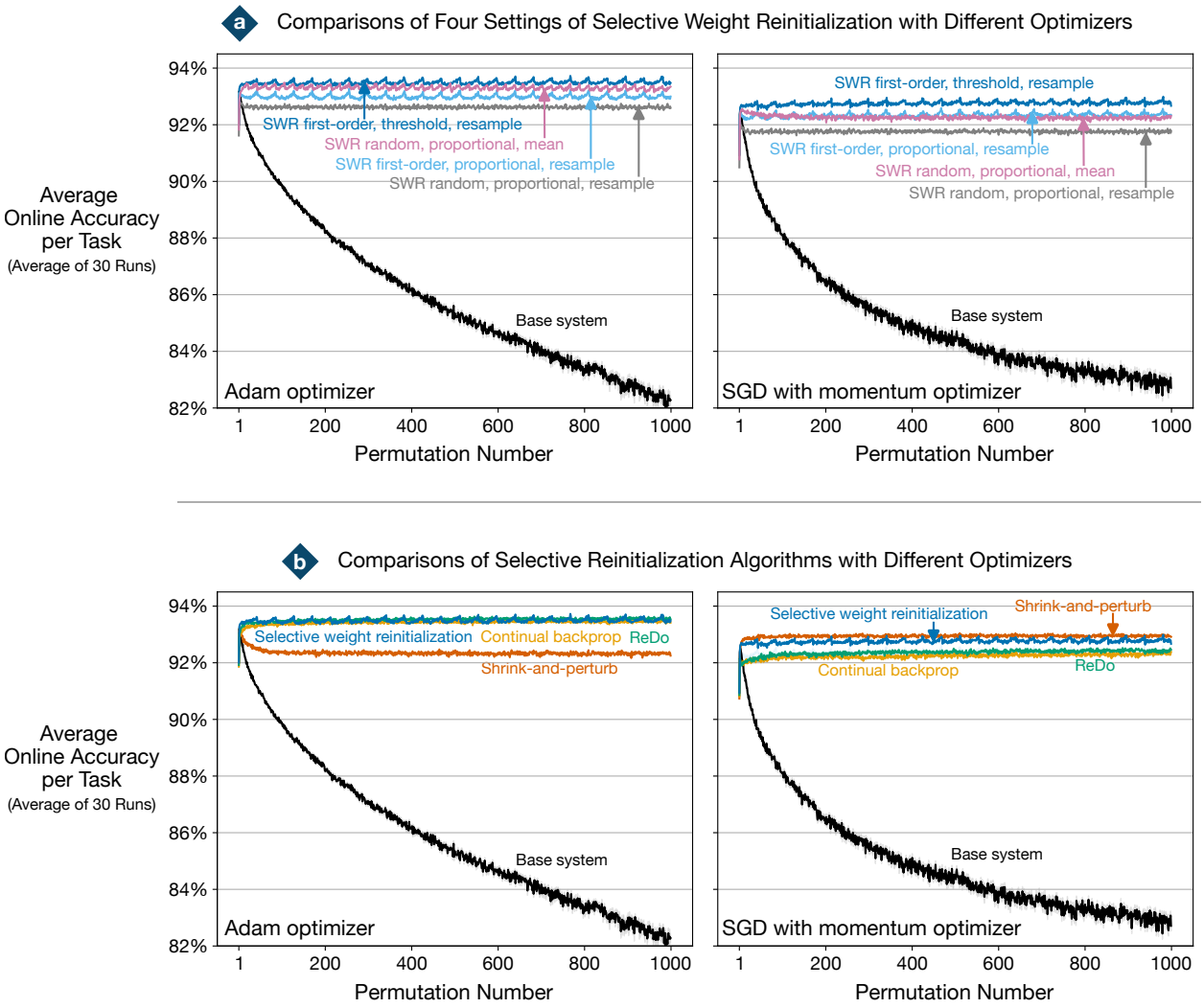

Figure 5.6: **(a)** Comparisons of different settings of selective weight reinitialization in learning systems trained with Adam or SGD with momentum. **(b)** Comparisons of selective reinitialization algorithms in learning systems trained with Adam and SGD with momentum. Each line corresponds to the sample average of 30 independent runs. The shaded regions indicate one standard error away from the sample average. All the selective reinitialization algorithms maintained plasticity with both optimizers. Hyperparameters are provided in Tables A.14 and A.15.

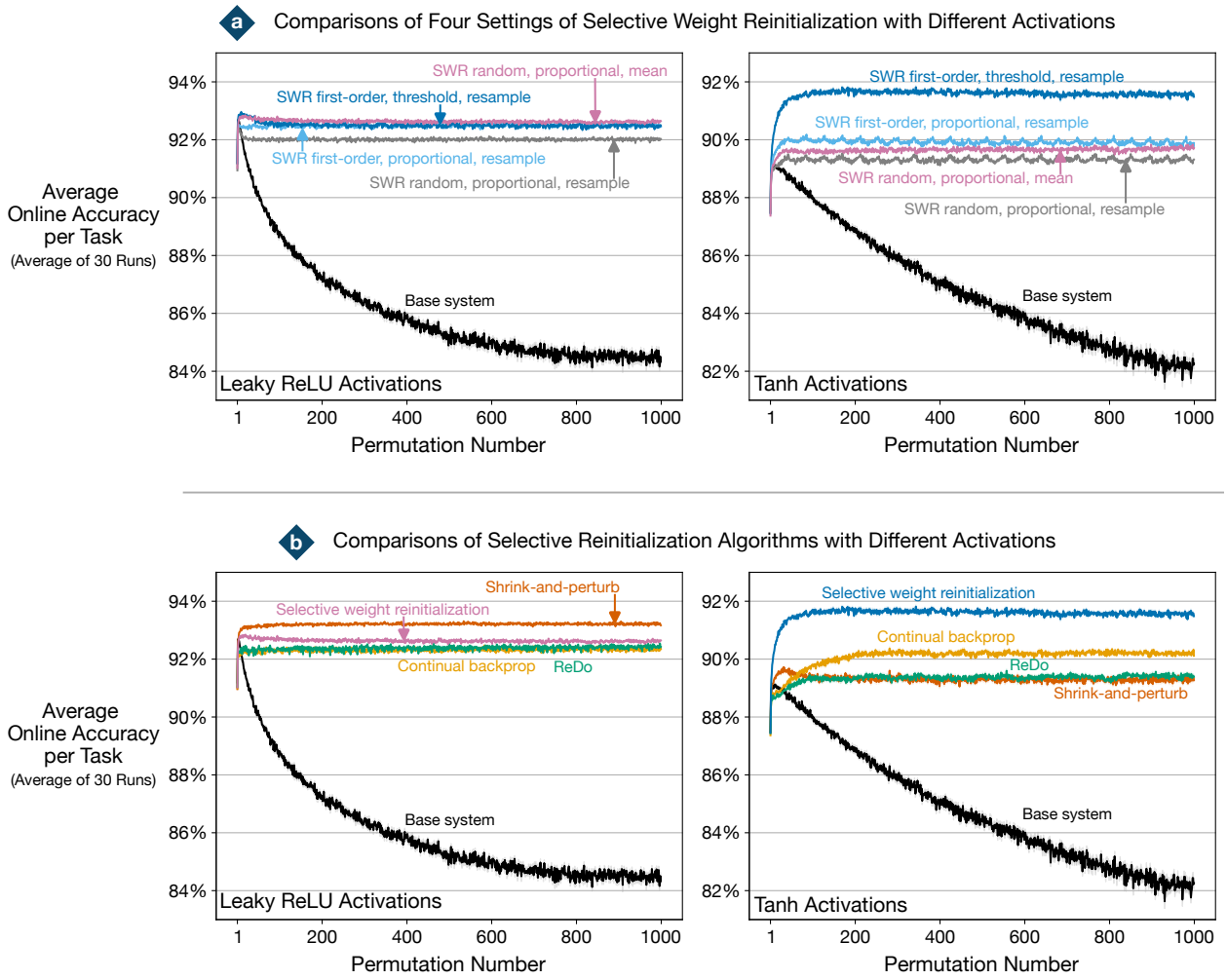

Figure 5.7: **(a)** Comparisons of different settings of selective weight reinitialization in learning systems with Leaky ReLU or Tanh activations. **(b)** Comparisons of selective reinitialization algorithms in learning systems with Leaky ReLU or Tanh activations. Each line corresponds to the sample average of 30 independent runs. The shaded regions indicate one standard error away from the sample average. The best setting of selective weight reinitialization with Leaky ReLU activations was random utility with proportional pruning and mean reinitialization, whereas with Tanh activations it was first-order utility with threshold pruning and resample reinitialization. Hyperparameters are provided in Tables A.16 and A.17.

respectively.

All four settings of selective weight reinitialization maintained plasticity with the two different activation functions (Figure 5.7a) In the Leaky ReLU system, the performance of selective weight reinitialization with random utility and mean reinitialization was higher than all the other settings. This is the first setting so far where random utility outperforms first-order utility with threshold

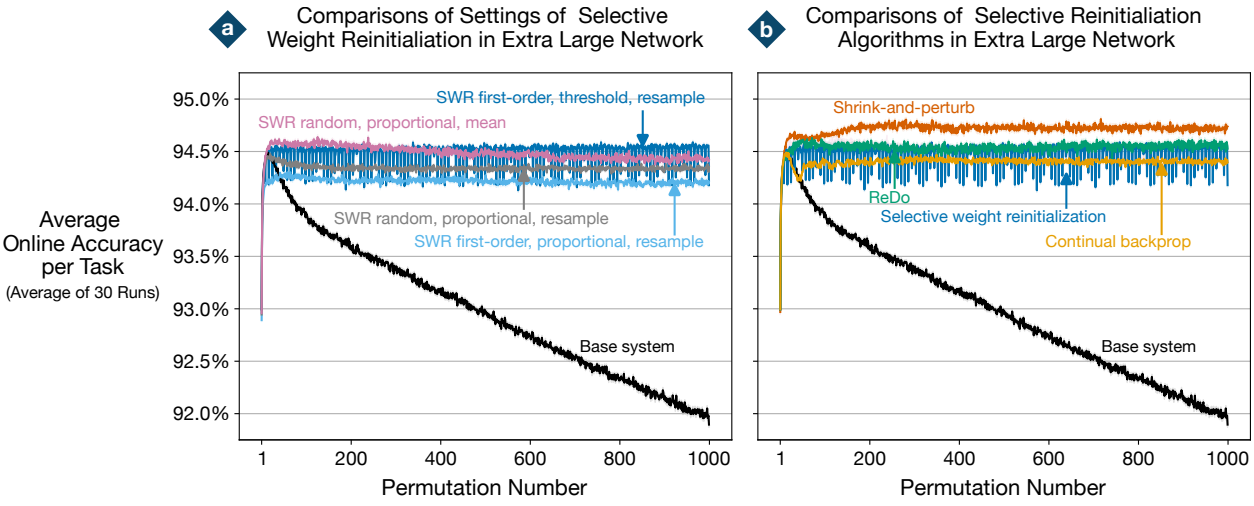

Figure 5.8: **(a)** Comparisons of different settings of selective weight reinitialization in the extra large network learning system. **(b)** Comparisons of selective reinitialization algorithms in the extra large network learning system. Each line corresponds to the sample average of 30 independent runs. The shaded regions indicate one standard error away from the sample average. Hyperparameter values provided in Table A.18.

pruning and resample reinitialization, highlighting the viability of the random utility measure. In the Tanh system, the setting with threshold pruning, first-order utility, and resample reinitialization had higher performance than other settings. Figure 5.7b shows comparisons of unit reinitialization algorithms against the best setting of selective weight reinitialization. In Leaky ReLU networks, unit and weight reinitialization yielded similar performance, whereas in Tanh networks, selective weight reinitialization achieved higher performance than continual backprop and ReDo.

The evaluations so far have shown that selective weight reinitialization is capable of maintaining plasticity in the same learning systems as unit reinitialization algorithms. For the last evaluation, let us examine how selective weight reinitialization scales with the network size compared to unit reinitialization algorithms. The base system in this evaluation is a ReLU network with three hidden layers and 1,000 units per layer— labelled *extra large network* in the plots.

All settings of selective weight reinitialization reduced the drop in performance observed in the base system (Figure 5.8a). However, not all settings showed stable performance. Random utility with mean reinitialization had a slight decrease in performance throughout the entire experiment, suggesting that the corresponding learning system may be losing plasticity. The other three settings of selective weight reinitialization maintained stable performance throughout the entire experiment. Figure 5.8b shows the performance of unit reinitialization algorithms and selective weight reinitialization with first-order utility, threshold pruning, and resample reinitialization. The performance of all three selective reinitialization algorithms stabilized around the same level. Moreover, the gap be-

tween the performance of selective reinitialization algorithms and shrink-and-perturb was smaller. These results suggest that as the network increases in size, the choice of a specific reinitialization algorithm is less relevant for maximizing performance.

A striking pattern in the results of the last evaluation is the drastic drops in performance in selective weight reinitialization (Figure 5.8). The observed pattern is once again the result of the reinitialization factor, which was set to 16,384, or approximately one reinitialization step every eight tasks. A consistent observation in the results so far is that, although selective weight reinitialization prevents the drop in performance due to loss of plasticity, its performance often shows several peaks and valleys. The hyperparameter settings of selective weight reinitialization influence the frequency of the peaks and valleys. However, the root cause of such irregularities in performance is that reinitialization impinges on the output of the network. Unit reinitialization may include mechanisms to reduce impingement; the output weights of new units are set to zero. Selective weight reinitialization has no such mechanisms.

The next part of this section compares weight against unit reinitialization based on how reinitialization impinges on performance. These comparisons are based on the results with the large network architecture in Figures 5.4a and 5.5a. Impingement is measured by comparing the loss on consecutive mini-batches before and after a reinitialization step. Specifically, let t be the time step when a reinitialization step occurs and $\{(\mathbf{x}_{t,i}, \mathbf{y}_{t,i})\}_{i=1}^{m}$ the mini-batch of data at such time step. The measure of impingement is defined as:

$$\text{Difference in Loss} \doteq \sum_{i=1}^{m} \ell(f_{\boldsymbol{\theta}_{t+1}}(\mathbf{x}_{t+1,i}), \mathbf{y}_{t+1,i}) - \sum_{j=1}^{m} \ell(f_{\boldsymbol{\theta}_t}(\mathbf{x}_{t,j}), \mathbf{y}_{t,j}). \tag{5.3}$$

Note that, as described in Algorithm 5, the evaluation of the loss happens before the reinitialization step.

Assuming there is no reinitialization or loss of plasticity, the quantity in the equation above would be negative if the loss is decreasing. When there is reinitialization, a negative difference in loss implies that reinitialization is not noticeably impinging on the loss. On the other hand, a positive difference in loss would imply that reinitialization is impinging on the loss, with higher positive values corresponding to higher impingement. Table 5.2 shows the average difference in loss per reinitialization step (*average difference in loss*), the percent of weights reinitialized per task (*weights reinitialized per task*), the percent of weights reinitialized per reinitialization step (*weights reinitialized per step*), and the average online accuracy. The average difference in loss is the relevant measure of impingement, whereas the other three quantities provide additional context about performance and the extent of reinitialization.

The summaries in Table 5.2 provide information about how reinitialization impinges on the loss. Continual backprop had a negative average difference in loss, meaning it did not noticeably impinge

Table 5.2: Summaries of the change in loss after a reinitialization step for different selective reinitialization algorithms. Each number is the average of 30 runs. The standard error is given in parentheses but is omitted in some columns for brevity, as it is too small to influence the comparisons.

Method	Average Difference in Loss	Weights Reinitialized per Task (%)	Weights Reinitialized per Step (%)	Average Online Accuracy (%)
SWR First-order, Threshold, Resample	0.0788 (0.0011)	48.58	49.78	92.79
SWR First-order, Proportional, Resample	0.1506 (0.0022)	19.52	40.00	92.42
SWR Random, Proportional, Resample	0.9904 (0.0034)	9.76	20.00	91.99
SWR Random, Proportional, Mean	0.1246 (0.0009)	9.77	5.00	92.62
Continual Backprop	−0.0022 (0.0002)	22.80	1.20	92.43
Redo	0.0044 (0.0003)	25.44	1.25	92.49

on the loss. ReDo had a positive average difference in loss, implying some amount of impingement. All the settings of selective weight reinitialization had considerably higher average difference in loss, implying their impigement on the loss was higher than unit reinitialization algorithms. However, there is a confounding variable that may be influencing the amount of impingement of each reinitialization algorithm: the percentage of weights reinitialized. All the selective weight reinitialization algorithms reinitialize a larger percentage of the weights per step than unit reinitialization algorithms. Likely, the high average difference in loss is not a consequence of the reinitialization method, but rather a result of the amount of reinitialization.

It is possible to control the amount of reinitialization in selective weight reinitialization with proportional pruning with an appropriate choice of hyperparameters. By setting the reinitialization factor to 0.012, we can ensure that the percentage of weights reinitialized for the step is 1.2%. In addition, we can set the reinitialization frequency to 100 to ensure that the percentage of weights reinitialized per task is 24%. In this way, we can closely match the amount of reinitialization observed in continual backprop and ReDo. Table 5.3 shows the same summaries as before, but with settings of selective weight reinitialization that reinitialize close to the same percentage of weights as unit reinitialization algorithms.

Even when controlling for the amount of reinitialization, selective weight reinitialization had a higher impingement on the loss than unit reinitialization algorithms. The average difference in loss for all four settings of selective weight reinitialization was positive and significantly larger than the difference in loss of unit reinitialization algorithms. Moreover, comparing settings of selective weight reinitialization to each other, mean reinitialization had a similar average difference in loss as resample reinitialization. In other words, mean reinitialization did not impinge less on the loss than resample reinitialization. Overall, selective weight reinitialization had a higher impingement on the loss than unit reinitialization algorithms.

The key takeaway from this section is that selective weight reinitialization often yields higher

Table 5.3: Summaries of the change in loss after a reinitialization step for different selective reinitialization algorithms, controlling for the amount of reinitialization. Each number is the average of 30 runs. The standard error is given in parentheses but is omitted in some columns for brevity, as it is too small to influence the comparisons. The average online accuracy marked with an asterisk corresponds to settings that failed to maintain plasticity.

Method	Average Difference in Loss	Weights Reinitialized per Task (%)	Weights Reinitialized per Step (%)	Average Online Accuracy (%)
SWR First-order, Proportional, Mean	0.1850 (0.0012)	24.00	1.20	88.84*
SWR First-order, Proportional, Resample	0.1793 (0.0004)	24.00	1.20	90.45*
SWR Random, Proportional, Mean	0.1851 (0.0003)	24.00	1.20	92.39
SWR Random, Proportional, Resample	0.1872 (0.0003)	24.00	1.20	91.90
Continual Backprop	−0.0022 (0.0002)	22.80	1.20	92.43
Redo	0.0044 (0.0003)	25.44	1.25	92.49

performance than unit reinitialization in learning systems based on fully-connected networks. However, selective weight reinitialization often showed higher impingement on the loss than unit reinitialization. Thus, although performance was higher for selective weight reinitialization, it was also more unstable.

5.5 The Effect of Selective Reinitialization on the Correlates of Plasticity Loss

This section revisits previous evaluations but studies how selective reinitialization impacts the different correlates of plasticity loss presented in Chapter 3. As a reminder, loss of plasticity is often accompanied by a decrease in the percent of dead units in the network, an increase in the magnitude of the weights of the network, a decrease in the average gradient magnitude, and a reduction in the stable rank of the output layer of the network—called the representation layer. We will start by revisiting evaluations where there was not a large difference between the effectiveness of selective reinitialization approaches. Then, we will move on to studying settings where there was a drastic difference in performance between different approaches.

For the first study, we revisit the large network setting presented in Figure 5.5a, where both unit and weight reinitialization were effective at maintaining plasticity. Figure 5.9 shows the performance of selective reinitialization algorithms along with the four correlates of plasticity loss. In the large network setting, all the reinitialization algorithms maintained plasticity and achieved similar levels of performance (Figure 5.9a). At the same time, selective reinitialization algorithms affected the correlates of plasticity in a similar manner. Specifically, they all prevented an increase in the percent of frozen units (Figure 5.9b), the increase of average weight magnitude in the network (Figure 5.9c), the decrease of average gradient magnitude of the loss (Figure 5.9d), and a decrease of the stable

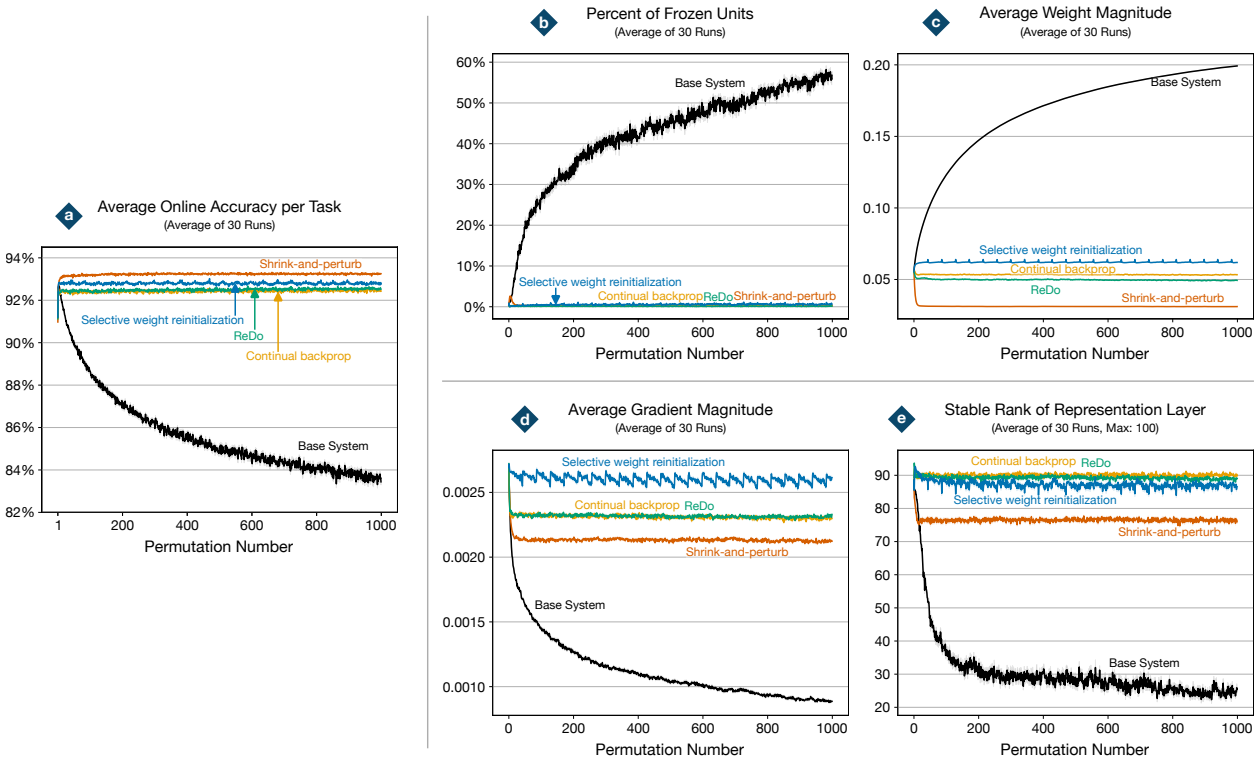

Figure 5.9: Correlates of plasticity loss in the large network setting. **(a)** Performance of selective reinitialization algorithms and baselines. **(b)** Percent of frozen units computed at the start of each task. **(c)** Average weight magnitude computed at the start of each task. **(d)** Average gradient computed on each mini-batch. **(e)** Stable rank of the representation layer. Each line corresponds to the sample average of 30 independent runs. The shaded regions indicate one standard error away from the sample average.

rank (Figure 5.9e).

Next, we move on to the Tanh network setting presented in the left pane of Figure 5.7b. In this setting, selective reinitialization was effective, but there were large differences in performance depending on the approach. Selective weight reinitialization with first-order utility with threshold pruning and resample reinitialization had the highest performance, followed by continual backprop and then ReDo (blue line in Figure 5.10a). Despite such a result, selective weight reinitialization had a decrease in stable rank (Figure 5.10e) and larger weight magnitude than ReDo and continual backprop (Figure 5.10c).

Lastly, we study the small network setting with layer norm presented in Figure 5.5c. In this setting, unit reinitialization approaches had reduced performance, whereas selective weight reinitialization maintained stable performance for the entire experiment (Figure 5.11a). Unit reinitialization algorithms had a steady decrease in performance and a decreasing gradient magnitude (Figure

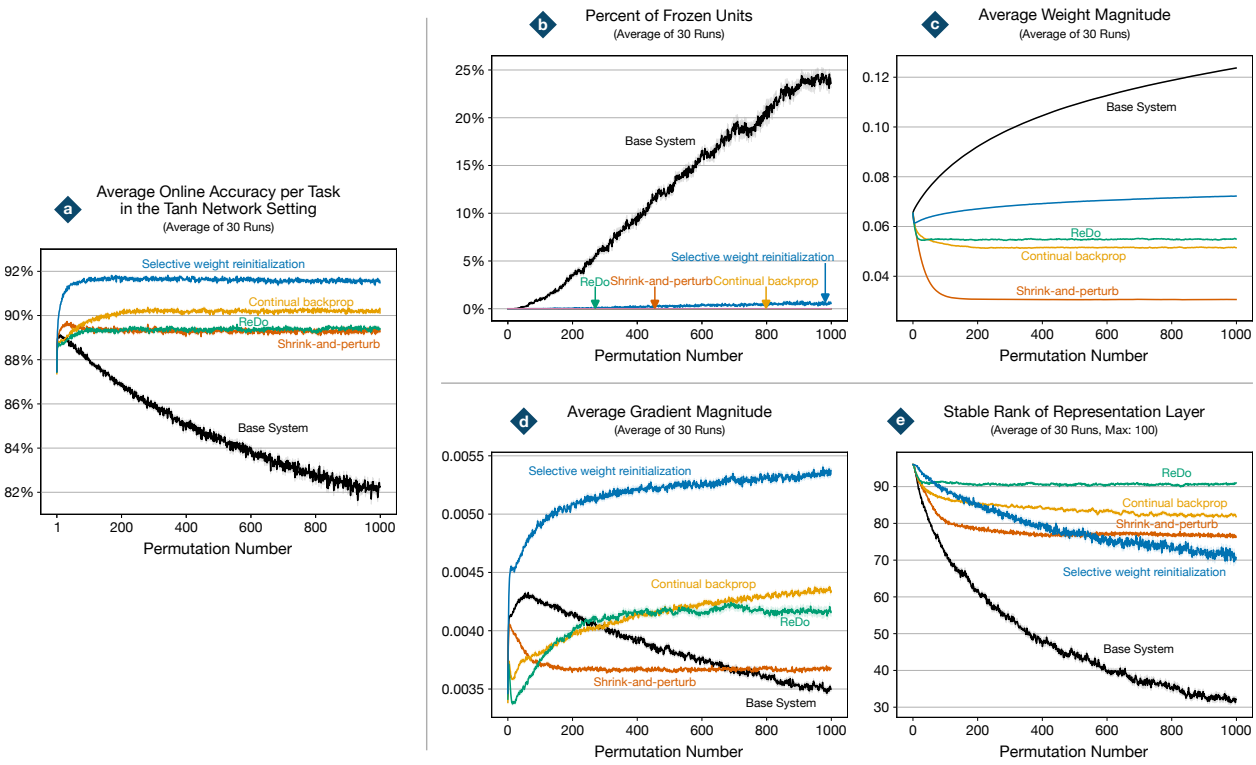

Figure 5.10: Correlates of plasticity loss in the Tanh network setting. **(a)** Performance of selective reinitialization algorithms and baselines. **(b)** Percent of frozen units computed at the start of each task. **(c)** Average weight magnitude computed at the start of each task. **(d)** Average gradient computed on each mini-batch. **(e)** Stable rank of the representation layer. Each line corresponds to the sample average of 30 independent runs. The shaded regions indicate one standard error away from the sample average.

5.11d) and stable rank (Figure 5.11e). Selective weight reinitialization also showed a decrease in gradient magnitude and stable rank, but nowhere as drastic as selective unit reinitialization methods.

The results in this section highlight that none of the correlates of loss of plasticity are perfect predictors of the phenomenon. In the large network setting, preventing plasticity loss was accompanied by a low percentage of frozen units, small weight magnitude, steady gradient magnitude, and a high stable rank. Nevertheless, that was not a consistent result across the three settings studied in this section. In the small network with layer norm setting, continual backprop suffered plasticity loss despite having a small percentage of frozen units (Figure 5.11b). In the same setting, selective weight reinitialization maintained plasticity despite having a decreasing gradient magnitude (Figure 5.11d). In the Tanh network setting, selective weight reinitialization maintained plasticity despite having a relatively large weight magnitude (Figure 5.10c) and a decreasing stable rank (Figure

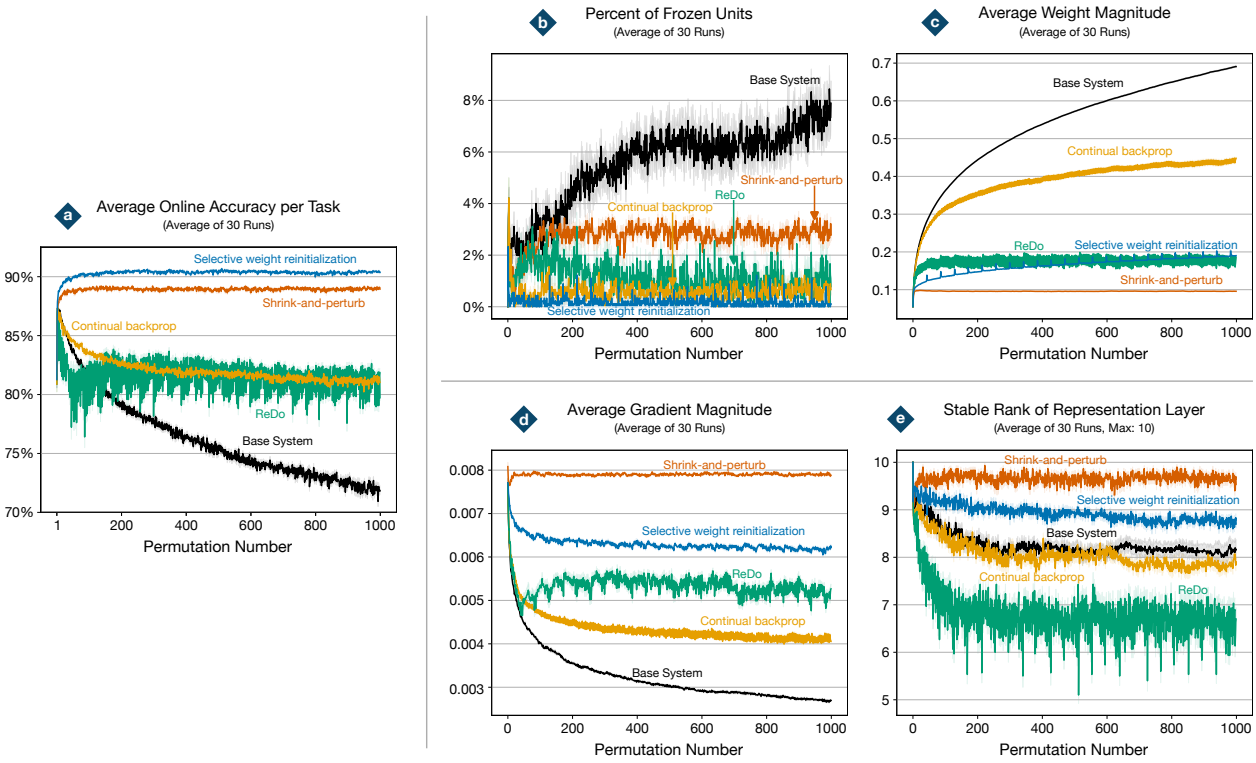

Figure 5.11: Correlates of plasticity loss in the Tanh network setting. **(a)** Performance of selective reinitialization algorithms and baselines. **(b)** Percent of frozen units computed at the start of each task. **(c)** Average weight magnitude computed at the start of each task. **(d)** Average gradient computed on each mini-batch. **(e)** Stable rank of the representation layer. Each line corresponds to the sample average of 30 independent runs. The shaded regions indicate one standard error away from the sample average.

5.10e).

The correlates of plasticity loss may be used as diagnostic tools. Their rapid deterioration could signal potential problems during learning. However, the four correlates of plasticity loss should not be interpreted as the root cause of the problem.

5.6 Future work on Selective weight Reinitialization and Alternative Approaches to Maintaining Plasticity

The results in this chapter demonstrate that selective weight reinitialization is a viable approach for maintaining plasticity in fully-connected neural networks. Moreover, in most of the evaluations presented in this chapter, selective weight reinitialization was competitive with or more effective

than selective unit reinitialization. Nevertheless, the results demonstrate two shortfalls of selective weight reinitialization that present opportunities for further improvements.

The first shortfall of selective weight reinitialization is that it suffers from persistent reinitialization. The term persistent reinitialization refers to the phenomenon where reinitialized weights are not given enough time to grow and, consequently, are selected repeatedly for reinitialization. The evidence for this phenomenon occurring is that a large proportion of the reinitialized weights are selected for reinitialization on consecutive reinitialization steps (see Table 5.1). An approach for addressing this issue is to introduce a maturity threshold, such as in continual backprop. A maturity threshold would allow only mature weights to be selected for reinitialization. The downside of this approach is that it would introduce a third hyperparameter to the algorithm. An alternative approach is to design a different reinitialization method that assigns higher values to new weights. Under both weight magnitude and first-order utilities, a large weight magnitude would increase the utility of the weight. Consequently, assigning high values to new weights would correspond to a higher initial utility, making it less likely that the corresponding weights would be selected again for reinitialization.

While assigning high values to new weights may reduce persistent reinitialization, such an approach may exacerbate the second shortfall of selective weight reinitialization: large impingement. The term impingement refers to how much a reinitialization step influences the output of the network and, consequently, the loss. A large amount of impingement may result in unstable learning, which could result in unstable performance. One possible way to decrease impingement is to reinitialize a smaller number of weights more frequently. Reinitializing a single weight every time step likely incurs less impingement than reinitializing 100 weights every 100 time steps. The downside of this approach is that it would likely result in higher persistent reinitialization. The most effective approach for solving impingement or persistent reinitialization is likely one that addresses both issues simultaneously.

Beyond the issues encountered in selective weight reinitialization, the algorithm can be improved by using different utility measures, pruning criteria, and reinitialization methods. Innovations and previous work on these three aspects of the algorithm were discussed in the previous chapter in the context of unit reinitialization. However, the same techniques can be applied in selective weight reinitialization.

An orthogonal line of future work is to modify selective weight reinitialization to train sparse networks. Selective weight reinitialization has many parallels to dynamic sparse training algorithms. Both types of algorithms include a utility measure, a pruning criterion, and a reinitialization method, and both modify the connections of the network with a certain frequency. However, dynamic sparse training algorithms maintain a target level of sparsity by pruning a proportion of active connections and reviving as many inactive connections (Mocanu et al., 2018; Evci et al.,

2020). To adapt selective weight reinitialization for sparse networks, one needs only to set a target sparsity level and apply pruning to active connections and reinitialization to inactive connections.

The next part of this section discusses alternative approaches to selective reinitialization for maintaining plasticity. Approaches for maintaining plasticity can be roughly categorized into four different types: architectural modifications, regularization, parameter perturbation, and selective reinitialization.

Architectural modifications propose changes to the network architecture to address the problem of loss of plasticity. Abbas et al. (2023) proposed the concatenated ReLU activation, a modification to ReLU units that activates on the positive and negative sides of the domain. Using concatenated ReLU was successful at preventing plasticity loss in deep reinforcement learning agents when learning from different Atari games in sequence. Nikishin et al. (2023) proposed a method for freezing old parameters and introducing new parameters to the network. Their approach, called plasticity injection, was successful at identifying plateaus in the performance of deep reinforcement learning agents in Atari games caused by plasticity loss and helped remediate them. Lyle et al. (2024a) proposed the use of layer norm on pre-activations along with weight rescaling to prevent plasticity loss. Their approach was used for improving the learning performance of deep reinforcement learning agents. Lee et al. (2024) proposed the hare-and-tortoise network, which maintains two copies of the network that learn at different timescales. Their architecture was successful at maintaining plasticity in continual supervised learning problems using residual networks and vision transformers.

Regularization approaches constrain the weights of the network to incentivize desirable properties in the solutions found by the stochastic optimization process. My co-authors and I showed that simple L2 regularization was effective at maintaining plasticity in the Permuted MNIST problem (Dohare et al., 2023). Kumar et al. (2024) proposed regenerative regularization, which is akin to L2 regularization but regularizes parameters towards their initial values instead of towards zero. Regenerative reinitialization was shown to maintain plasticity in a wider variety of continual supervised learning problems than standard L2 regularization. Elsayed et al. (2024) proposed clipping the weights so they remain below a certain prespecified threshold. Their approach prevented loss of plasticity in continual supervised learning problems and the policy collapse problem encountered in deep reinforcement learning. Chung et al. (2024) proposed a regularization technique for incentivizing orthogonal weight matrices in the network. Their regularization approach prevented the decrease in stable rank in neural networks associated with plasticity loss and improved the performance of deep reinforcement learning agents. Lastly, Lewandowski et al. (2025) proposed a spectral regularization technique that regularizes weight matrices of each layer so that their largest singular value is close to one. Their method was capable of preventing plasticity loss in residual networks and vision transformers trained on class incremental tasks.

Parameter perturbation approaches introduce noise to the parameters of the network. Ash and

Adams (2020) proposed the algorithm shrink-and-perturb, which has been the subject of many evaluations in this and the previous two chapters. The authors proposed the algorithm to prevent the loss in performance when the network is pre-trained on a small portion of the dataset before being trained on the full dataset. At the time, such a decrease in performance was not recognized as an instance of loss of plasticity. Nevertheless, shrink-and-perturb is highly effective at preventing plasticity loss in multiple settings (Dohare et al., 2024; Kumar et al., 2024). Elsayed and Mahmood (2024) proposed an algorithm for adding parameter noise to the weights of the network proportional to the utility of the weights. Their algorithm was effective in maintaining plasticity in continual supervised learning problems and preventing policy collapse in deep reinforcement learning agents.

Selective reinitialization methods sporadically reinitialize parts of the network to regenerate the initial conditions that facilitated training. Related work on selective reinitialization at the level of the unit is presented in the previous chapter. My co-authors and I were the first to study selective reinitialization at the level of the weights (Hernandez-Garcia et al., 2025). In the corresponding paper, we showed that selective weight reinitialization is capable of maintaining plasticity in several continual supervised learning problems and in a wide variety of architectures. In parallel, Hofmann et al. (2025) proposed an architecture that includes a procedure called weight rejuvenation. Weight rejuvenation takes a probabilistic approach to reinitializing weights in a network: weights with small magnitude have a high probability of being reinitialized. Weight rejuvenation, along with other biologically inspired modifications, was shown to increase accuracy in image classification problems with fewer parameter updates. While not designed to prevent plasticity loss, the experiments in this chapter suggest that weight rejuvenation may also be effective at maintaining plasticity.

The four approaches to preventing plasticity loss are not mutually exclusive from each other. In fact, the combination of many of these approaches is often more effective than their individual use. For example, in an ant locomotion reinforcement learning task, the combination of continual backpropagation and L2 regularization was shown to be more effective at maintaining plasticity than either method alone (Dohare et al., 2024). Similarly, Lee et al. (2023) combined L2 regularization, layer normalization, layer reinitialization, and concatenated ReLU activations to prevent plasticity loss in deep reinforcement learning agents. The sample efficiency of deep reinforcement learning agents can be improved by increasing the number of parameter updates per environment step, a quantity known as the replay ratio. Large replay ratios improve sample efficiency of agents but have also been observed to result in poor performance due to loss of plasticity (D'Oro et al., 2023). The combination of several plasticity-preserving approaches proposed by Lee et al. (2023) resulted in an agent more robust to large replay ratios than any of the individual approaches.

5.7 Discussion and Conclusion

This chapter presented an alternative approach for implementing selective reinitialization that performs reinitialization at the level of the weights. The corresponding algorithm is called selective weight reinitialization. Selective weight reinitialization has three key components: the utility measure, the pruning criterion, and the reinitialization method.

The first evaluations in this chapter presented the performance of different choices of each of the key components of selective weight reinitialization. These initial evaluations considered three different utility measures, two different pruning criteria, and two different reinitialization methods. Not every setting of selective weight reinitialization maintained plasticity. None of the settings with weight magnitude utility maintained plasticity in the Permuted MNIST problem. All settings using random utility maintained plasticity. Moreover, threshold and proportional pruning had equivalent performance when using random utility. Lastly, first-order utility was effective at maintaining plasticity with both pruning criteria as long as the algorithm used resample reinitialization.

After the initial evaluation, the chapter presented evaluations comparing the effectiveness of reinitializing weights against units for maintaining plasticity. Specifically, the evaluations compared the most promising settings of selective weight reinitialization against continual backprop and ReDo. These evaluations included comparisons with different network sizes, different activation functions, networks with layer normalization, and different optimizers. Selective weight reinitialization was equally or more effective at maintaining plasticity than continual backprop and ReDo. Nevertheless, in large networks, the gap in performance between reinitialization algorithms was drastically reduced, which may imply that with sufficiently large networks, these algorithms may have comparable performance.

While being effective at maintaining plasticity, selective weight reinitialization faces two challenges that could be addressed in future work. First, reinitializing weights has a higher effect, or impingement, on the performance of the learning system than reinitializing units. As a consequence of this large impingement, the performance of selective weight reinitialization was unstable. Second, reinitializing weights often resulted in the same subset of weights being reinitialized, a phenomenon I called persistent reinitialization. Persistent reinitialization was associated with a failure to maintain plasticity and is likely caused by weights not being given enough time to grow after being reinitialized.

Lastly, let us address the question of which setting of selective weight reinitialization is most effective. After all the evaluations presented in this chapter, the setting that consistently outperformed all other settings was the one using first-order utility with threshold pruning and resample reinitialization. Such a setting of selective weight reinitialization was only outperformed by random utility with mean reinitialization when using a network with 100 Leaky ReLU units per layer (See

Figure 5.7b). Unless otherwise specified, the following chapters present results using first-order utility with threshold pruning and resample reinitialization.

The evaluations presented in this and the previous chapters have only used fully-connected networks. The following chapters study loss of plasticity in increasingly complex architectures and the effectiveness of selective reinitialization in such architectures.

Chapter 6

Loss of Plasticity in Convolutional Networks

The previous chapters established that plasticity loss is a widespread phenomenon and that selective reinitialization is a suitable approach for preventing it. However, the studies were limited to fully-connected networks, which differ from the contemporary systems used in practice. A remaining open question is whether modern architectures are also susceptible to plasticity loss and whether selective reinitialization remains effective.

This chapter continues the demonstrations of plasticity loss, extending them to systems based on convolutional neural networks. The experiments are performed using a continual binary classification problem based on the ImageNet dataset. The results establish that systems based on convolutional neural networks are also susceptible to plasticity loss.

After establishing the persistence of the phenomenon, the chapter demonstrates the effectiveness of selective reinitialization at preventing it. The results show that continual backpropagation remains a robust approach for maintaining plasticity, while ReDo and selective weight reinitialization succeed but exhibit learning instabilities. Finally, I show that combining regularization with selective reinitialization eliminates instabilities and improves performance.

This work extends collaborative research published in Nature (Dohare et al., 2024) by evaluating ReDo and selective weight reinitialization on an alternative downsampled ImageNet dataset and convolutional architecture.

6.1 Continual Binary Classification in Tiny ImageNet

The ImageNet dataset is a popular benchmark for computer vision models (Deng et al., 2009). Because of its large number of classes, the number of examples, and the high dimensionality of the data, the dataset is suitable for testing the capabilities of state-of-the-art deep learning systems. This chapter repurposes the ImageNet dataset to design a continual learning problem called *Continual ImageNet*. The Continual ImageNet problem is the testbed used in this chapter to study loss of plasticity in learning systems based on convolutional network architectures.

The Continual ImageNet problem consists of a sequence of binary classification problems based on images from the ImageNet dataset. At the start of the problem, a learning system is tasked with distinguishing between two types of objects. Once performance has plateaued, a new pair of objects is selected to create a new binary classification problem, also called a new task. This process repeats for thousands of tasks, which is illustrated in Figure 6.1. After each task, the network's output layer is reset to zero, which can be viewed as initializing new output units for the new classes.

In the Continual ImageNet problem, a learning system is trained on thousands of tasks to assess its ability to learn from non-stationary data. To evaluate how well a system is learning, its accuracy is measured on a held-out set—*the test set*. For each task, the learning system does several passes through the dataset, also known as epochs. The system's performance on each task is the test accuracy measured after each epoch and averaged across all epochs in the task—the *average test accuracy*. This measure of performance is analogous to the average empirical loss discussed in Chapter 3 (see equation 3.3).

The classes of each task in the Continual ImageNet problem are randomly sampled, so, in expectation, the tasks are equally difficult. Assuming the classes never repeat, any changes in performance could only be attributed to the system's ability to learn. What would be the trend in average performance as the system learns from new tasks? If a system maintains its ability to learn from new tasks, we would expect average performance to remain mostly flat. Alternatively, since images from different classes share common attributes, information learned from early tasks may facilitate learning in later tasks. In such a case, we would expect the average performance to increase as the system learns. On the other hand, if the system loses plasticity, we would expect its average performance to decrease as the number of tasks increases. Consequently, we would observe a decreasing trend in the average performance as the number of tasks increases.

Before moving on to studying the capabilities of deep learning systems in the Continual ImageNet problem, it is necessary to provide details about the dataset to guarantee the reproducibility of the results. The version of ImageNet used in this chapter is known as Tiny ImageNet (Le and Yang, 2015) and is available for download through the Hugging Face datasets python package

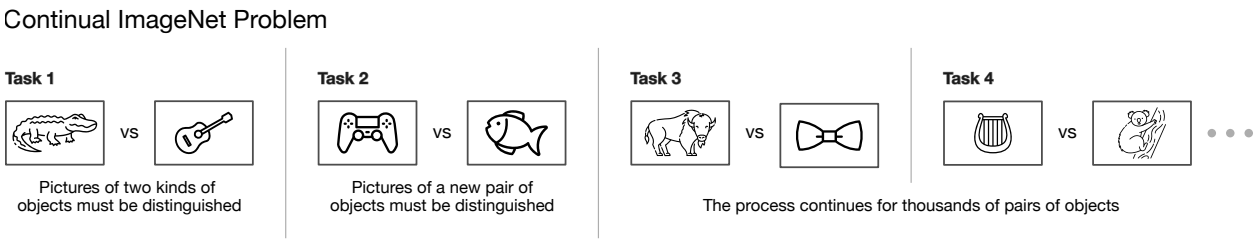

Figure 6.1: Description of the Continual ImageNet problem. To be successful in this problem, a learning system must maintain its ability to learn for thousands of tasks.

(Lhoest et al., 2021). The dataset consists of 200 classes, with 500 training images per class and 50 test images per class. The images are 64 pixels tall and 64 pixels wide. Most images consist of three channels corresponding to the colours red, green, and blue. However, some of the images only have a single channel. To handle such images, the single channel is copied three times to create a black-and-white image. Thus, after handling single-channel images, the images have dimensions $64 \times 64 \times 3$. As another preprocessing step, each channel in each image is normalized by subtracting the sample mean and dividing by the standard deviation computed from the entire training set.

Note that the Tiny ImageNet dataset contains only 200 classes. When using Tiny ImageNet to create the Continual ImageNet problem, classes are randomly assigned to 100 different pairs. Once a learning system is trained for 100 tasks, the classes are randomized again into 100 new pairs. Consequently, from the learning system's perspective, only the first 100 tasks involved brand-new classes. Afterwards, the system starts learning from previously observed images and classes. Thus, even if learning systems lose plasticity, the drop in performance may not be as severe due to remembering. This is a flaw in the experimental design, but one I consciously chose to incur because Tiny ImageNet is a well-studied downsampled version of the ImageNet dataset. To reduce the amount of remembering, experiments use 250 epochs per task to incentivize systems to completely override information about the previous task.

6.2 Loss of Plasticity in Continual ImageNet

This section investigates whether learning systems based on convolutional neural networks exhibit plasticity loss in the Continual ImageNet problem. Each experiment in this section trains a convolutional neural network on 2,000 tasks of the Continual ImageNet problem. Networks are trained for 250 epochs per task using stochastic gradient descent (SGD) with momentum of 0.9, minimizing cross-entropy loss with a mini-batch size of 100.

The goal of the first evaluation is to examine whether convolutional layers lose plasticity. This

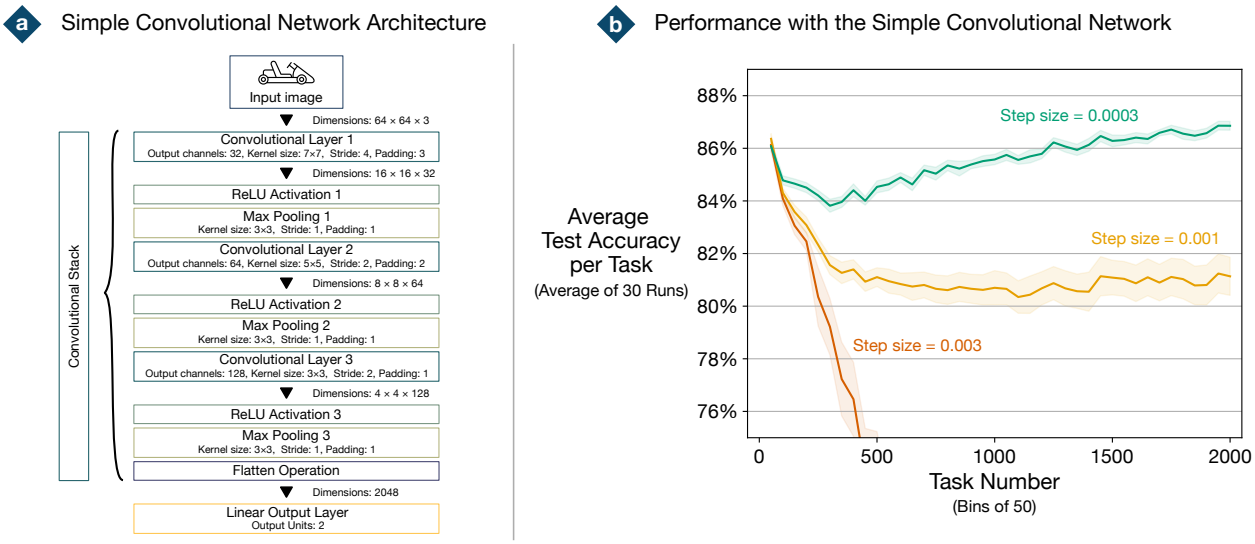

Figure 6.2: **(a)** Simple convolutional network architecture. **(b)** Performance with the simple convolutional network using different step-size values for the SGD with momentum optimizer. Each line is the average of 30 runs, while shaded regions correspond to one standard error away from the average. Tasks are binned in groups of 50. Hyperparameter values are provided in Table A.19.

initial evaluation utilizes a network comprising a stack of three convolutional layers. A ReLU activation and a max pooling operation follow each convolutional layer. The output of the last convolutional layer is flattened and connected to a linear output layer with two units, one for each class in the task. The weight matrices in hidden layers were initialized using Kaiming normal initialization, whereas the bias terms and the output weight matrix were initialized to zero. For further details on the network architecture, see Figure 6.2a. Let us refer to this first network architecture as *simple convolutional network*. Figure 6.2b shows the performance of the simple convolutional network with three different step-size values for SGD.

The performance of the three learning systems decreased during the first 300 tasks (Figure 6.2). After the first 300 tasks, once learning systems began experiencing repeated classes, the performance of learning systems with larger step-sizes continued to decrease. On the other hand, the performance of the system with a step-size of 0.0003 began to increase after 300 tasks.

The next evaluation examines whether systems with a mix of convolutional and fully-connected layers exhibit plasticity loss. The network in this evaluation has the same architecture as before, but includes two fully-connected layers after the convolutional stack. Let us refer to this network as the *mixed convolutional network*. The details of the architecture are shown in Figure 6.3a.

The learning systems based on the mixed convolutional network showed different trends in

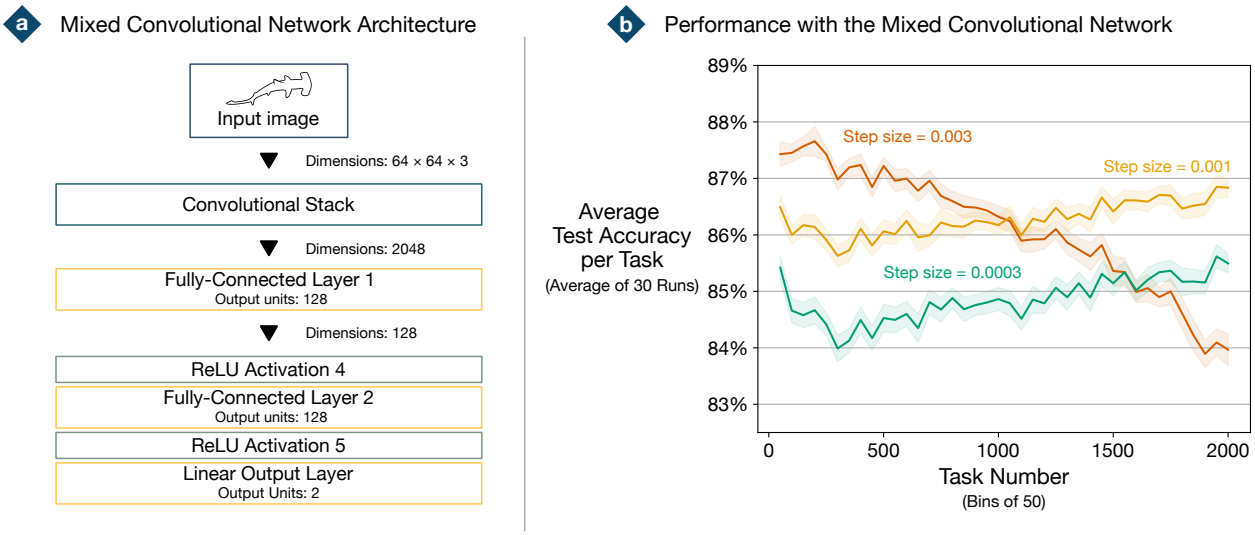

Figure 6.3: **(a)** Mixed convolutional network architecture. **(b)** Performance with the mixed convolutional network using different step-size values for the SGD with momentum optimizer. Each line is the average of 30 runs, while shaded regions correspond to one standard error away from the average. Tasks are binned in groups of 50. Hyperparameter values are provided in Table A.19.

performance based on the step-size (Figure 6.3b). Systems with step-sizes of 0.001 and 0.0003 showed an initial decrease in performance over the first 300 tasks, followed by a steady increase. Once again, the performance increase began after the system experienced repeated classes. On the other hand, the system with a step-size of 0.003 initially showed an increase in performance, followed by a steady decrease.

The results of these two evaluations are reminiscent of the results in Permuted MNIST in Chapter 3. In Permuted MNIST, networks with a large step-size experienced a steeper decrease in performance compared to networks with a small step-size (see Figure 3.2a). Nevertheless, all the networks experienced some degree of plasticity loss. In the Continual ImageNet problem, a similar result is observed. All the networks experienced a decrease in performance, but those trained with a smaller step-size suffered a less severe effect and eventually recovered.

The next step in the evaluations in this section involves finding suitable baselines for methods that maintain plasticity in Continual ImageNet. In the previous chapters, L2 regularization and shrink-and-perturb were effective in maintaining plasticity across a wide range of learning systems based on fully-connected networks. Thus, they are promising modifications to the base systems that may prevent plasticity loss in Continual ImageNet.

The base systems for the next set of evaluations are the simple convolutional network with a step-size of 0.001 and the mixed convolutional network with a step-size of 0.003. These two step-size

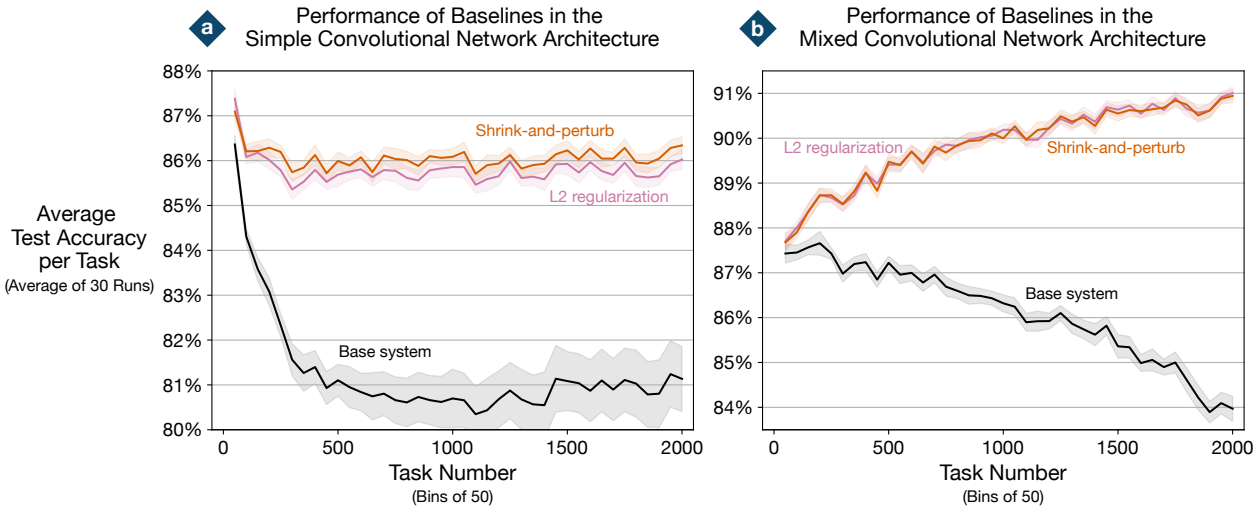

Figure 6.4: **(a)** Performance with the simple convolutional network when augmented with L2 regularization and shrink-and-perturb. **(b)** Performance with the mixed convolutional network when augmented with L2 regularization and shrink-and-perturb. Each line is the average of 30 runs, while shaded regions correspond to one standard error away from the average. Tasks are binned in groups of 50. Hyperparameter values are provided in Table A.19.

values were selected because they resulted in high initial performance but led to severe loss of plasticity. L2 regularization and shrink-and-perturb were added to these to base learning systems, but the networks were trained using SGDW with momentum. SGDW is a variant of SGD that excludes the regularization term from the moving average of the gradient, which has been reported to result in more stable performance (Loshchilov and Hutter, 2019). The L2 regularization parameter was tuned to maximize the area under the curve of the average test accuracy. Shrink-and-perturb used the same regularization factor, but the variance of the noise was tuned. Hyperparameter tuning followed the same procedure as before: a wide search using a single run followed by a narrow search with multiple runs, in this case five runs per hyperparameter combination.

There was a significant difference in results across network architectures. In the simple convolutional network, L2 regularization mitigated plasticity loss but did not completely remove the problem (Figure 6.4a). On the other hand, in the mixed convolutional network, L2 regularization not only removed the problem but also enabled a steady increase in performance (Figure 6.4b). In either case, the performance of shrink-and-perturb was not much different from that of L2 regularization.

The results in this section demonstrate that learning systems based on convolutional neural networks are susceptible to plasticity loss in the Continual ImageNet problem. Using a small step-size for the optimizer helped mitigate the plasticity loss, but did not eliminate it. This evidence establishes that plasticity loss extends to convolutional neural networks. Using L2 regularization

and shrink-and-perturb mitigated the problem in the simple convolutional network and completely prevented it in the mixed convolutional network.

6.3 Maintaining Plasticity in Continual ImageNet With Selective Reinitialization

This section examines the effectiveness of selective reinitialization at preventing plasticity loss in learning systems based on convolutional networks. Both approaches to selective reinitialization can be applied to convolutional networks. For weight reinitialization, the application is straightforward since the algorithm does not need to consider the network's connectivity pattern. On the other hand, unit reinitialization approaches need to define what constitutes a unit in a convolutional layer.

For the purpose of applying unit reinitialization, the basic unit is a convolutional channel (see Figure 4.8b for an illustration). Since channels are matrices, the computation of utility measures is slightly different. The average activation of a unit in a convolutional layer is defined as the average over the number of samples in the mini-batch and the channel's width and height. Such an operation reduces the channel to a single scalar number, which can be used to compute the utility measures defined in Chapter 4. In addition to how utilities are computed, one must also decide whether to base utilities on the ReLU activations or the max pooling layer outputs. In this chapter, utilities for unit reinitialization algorithms are computed based on the output of the max pooling layer. Nevertheless, using the ReLU activations' outputs to compute utilities yielded similar performance during hyperparameter tuning.

The first experiment in this section examines the effectiveness of selective reinitialization algorithms when augmenting a simple convolutional network. The base system is the simple convolutional network with a step-size of 0.001. For selective unit reinitialization, ReDo with activation utility or continual backprop with contribution utility augments the base system. Both ReDo and continual backprop used hemi-reinitialization. For selective weight reinitialization, the setting using first-order utility with threshold pruning and resample reinitialization is used to augment the base system. For ReDo, the reinitialization frequency and threshold were tuned, for continual backprop, the maturity threshold and replacement rate, and for selective weight reinitialization, the reinitialization frequency and factor.

All the selective reinitialization algorithms prevented the drop in performance observed in the base system (Figure 6.5a). In other words, they all maintained plasticity. Nevertheless, performance varied depending on the approach. Selective weight reinitialization maintained plasticity, but its performance was lower than that of unit reinitialization algorithms. Among the two unit reinitialization algorithms, continual backprop achieved higher performance than ReDo. However,

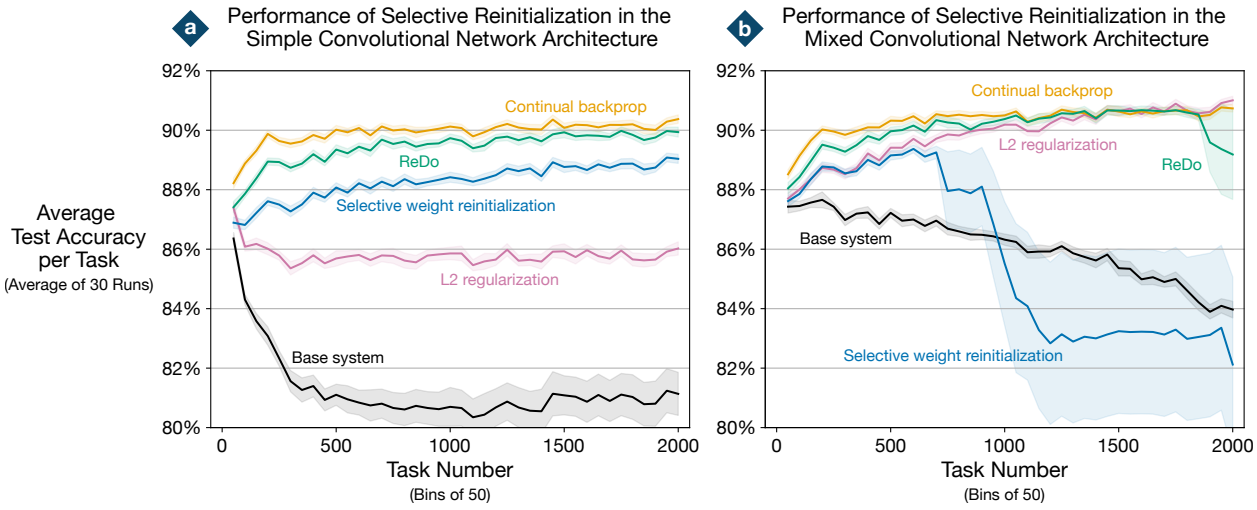

Figure 6.5: **(a)** Performance with the simple convolutional network when augmented with selective reinitialization approaches. **(b)** Performance with the mixed convolutional network when augmented with selective reinitialization approaches. Each line is the average of 30 runs, while shaded regions correspond to one standard error away from the average. Tasks are binned in groups of 50. Hyperparameter values are provided in Table A.20.

the gap in performance between unit reinitialization algorithms narrowed as the number of tasks increased.

The results obtained with the simple convolutional network contrast with those obtained with fully-connected networks in the previous chapter. In learning systems based on fully-connected networks, selective weight reinitialization often achieved higher performance than unit reinitialization algorithms. It may be that selective weight reinitialization has an advantage over unit reinitialization algorithms when a convolutional network includes fully-connected layers.

The following experiment examines the performance of selective reinitialization algorithms with the mixed convolutional network architecture. The base system was the mixed convolutional network with a step-size of 0.003. L2 regularization is added as a baseline.

The results with the mixed convolutional network are drastically different from the results with the simple convolutional network (Figure 6.5b). Continual backprop was still capable of maintaining plasticity and achieving good performance. However, the gap between L2 regularization and continual backprop was smaller. ReDo maintained plasticity as well, and the difference between its performance and continual backprop's was smaller. However, ReDo had unstable learning in the last few tasks, as evidenced by the wide shaded regions. Lastly, selective weight reinitialization initially maintained plasticity, but its performance became increasingly unstable over time.

The wide error bars in the performance of ReDo and selective weight reinitialization are due

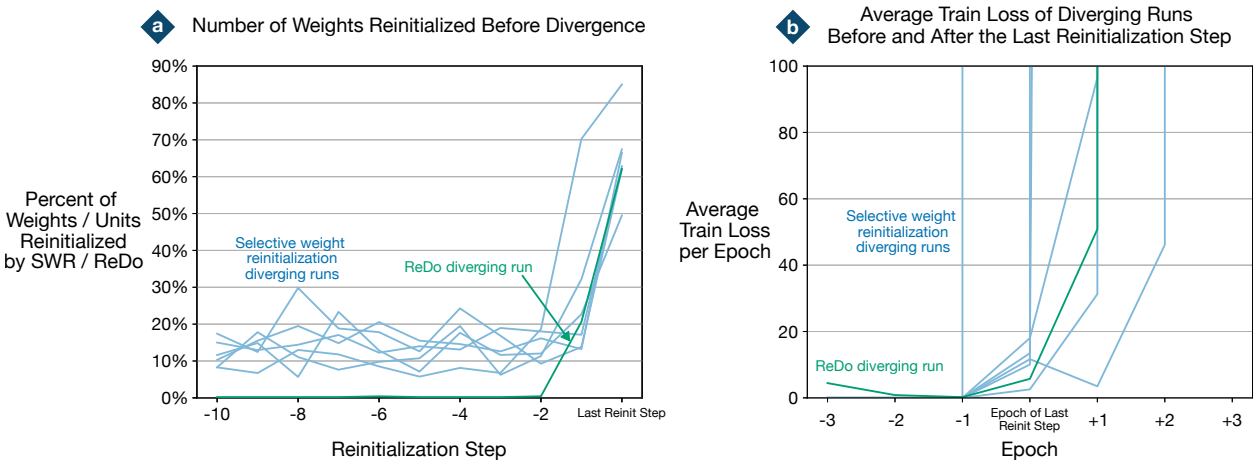

Figure 6.6: **(a)** Number of units reinitialized by ReDo and number of weights reinitialized by selective weight reinitialization before divergence. **(b)** Average train loss per epoch of learning systems using ReDo or selective weight reinitialization. Each line corresponds to a single run that eventually diverged. Divergence was preceded by a sudden increase in the number of reinitialized weights or units, which was followed by a steady increase in loss.

to the optimization process diverging in some runs. Specifically, six runs of selective weight reinitialization and one run of ReDo diverged. The reason for this divergence was that the selective reinitialization algorithms suddenly reinitialized a large portion of the network. The large proportion of reinitialized weights resulted in a significant change in loss, which subsequently increased until it diverged. Figure 6.6a shows the number of reinitialized weights or units for each run that diverged leading up to the last reinitialization step. In all the runs, divergence was preceded by an increase in the number of reinitialized weights or units. The increase in reinitialization was followed by an increase in loss, which eventually overflowed due to extremely large values (Figure 6.6b).

The reason ReDo and selective weight reinitialization suddenly reinitialized a large portion of the weights is that they both used threshold pruning. As a reminder, threshold pruning removes all the weights that fall below a threshold that is a function of the average utility of the weights in the matrix. Threshold pruning may prune all but one unit or weight if an extreme outlier drastically affects the sample average. Consequently, outliers may cause a sudden increase in reinitialized weights, such as the ones observed in Figure 6.6a.

There are many ways to prevent such a drastic increase in the number of weights reinitialized. The simplest approach is to use proportional pruning for weight reinitialization or continual backprop for unit reinitialization. Figure 6.7a shows the previous results but with two different settings for selective weight reinitialization with proportional pruning: a setting with random utility and mean reinitialization (abbreviated *SWR: random, proportional, mean* in Figure 6.7a) and a setting with first-order utility and resample reinitialization (abbreviated *SWR: first-order, proportional, re-*

sample). Selective weight reinitialization with random utility and mean reinitialization maintained plasticity without any diverging runs, though it did not match the performance of L2 regularization or continual backprop. On the other hand, selective weight reinitialization with first-order utility and resample reinitialization did not improve upon the performance of the base system. Notably, during the hyperparameter search, the setting with first-order utility also showed divergence for large reinitialization factors, limiting it to only small reinitialization amounts.

While proportional pruning is a simple fix, it eliminates threshold pruning, which often yielded the best performance in the evaluations in previous chapters. Therefore, exploring other modifications that enable threshold pruning remains worthwhile. A simple modification for threshold pruning is to use the median instead of the mean. The median is more robust to outliers, so extreme utility values would be less likely to cause a large portion of the network to be reinitialized. While promising, a comprehensive study of this threshold pruning modification warrants thorough experimentation beyond the scope of this chapter. An alternative approach is to use L2 regularization to prevent weights from growing too large and prevent learning instabilities.

The following experiment studies the effectiveness of combining L2 regularization and selective reinitialization. As alluded to in previous chapters, combining several approaches to maintain plasticity is often more effective than any single approach. By studying the performance of the combination of L2 regularization and selective reinitialization, we may gain insights about how the two approaches complement each other. The following experiment evaluates the combination of L2 regularization with ReDo, continual backprop, and selective weight reinitialization. ReDo was instantiated with activation utility, continual backprop with contribution utility, and selective weight reinitialization with first-order utility with threshold pruning and resample reinitialization (labelled *SWR: first-order, threshold, resample* in Figure 6.7b). The regularization factor was set to the same value as the L2 regularization baseline, but the hyperparameters of each selective reinitialization algorithm were tuned.

Combining L2 regularization with selective reinitialization resulted in a performance improvement (Figure 6.7b). For continual backprop, the improvement was modest, but for ReDo and selective weight reinitialization, the improvements were considerable. Not only did none of the runs of ReDo or selective weight reinitialization diverge, but their performance also increased to a level close to that of continual backprop. L2 regularization not only stabilized learning but also reduced the performance gap between selective reinitialization algorithms. Overall, combining selective reinitialization with L2 regularization yielded improved performance compared to L2 regularization alone.

The experiments in this section demonstrated that selective reinitialization approaches were effective in maintaining plasticity in the two network architectures of Continual ImageNet. These results suggest that the effectiveness of selective reinitialization extends to convolutional network

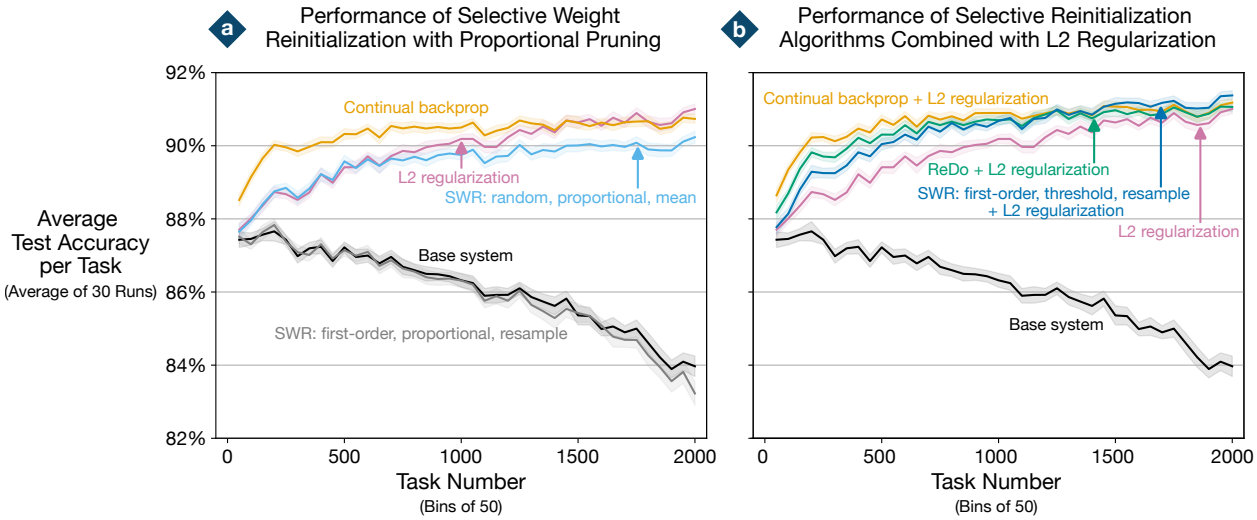

Figure 6.7: Experiments with the mixed convolutional network learning system augmented with **(a)** selective weight reinitialization (SWR) with proportional pruning and **(b)** selective reinitialization algorithms combined with L2 regularization. Each line corresponds to the average of 30 independent runs. The shaded region corresponds to the standard error. Using proportional pruning with SWR prevented the divergence observed with threshold pruning. Using L2 regularization also prevented divergence and reduced the performance gap between different reinitialization algorithms. Hyperparameters are provided in Table A.21.

architectures and mixed networks that combine convolutional and fully-connected layers. Moreover, selective reinitialization was more effective than L2 regularization and shrink-and-perturb in Continual ImageNet. Nevertheless, selective reinitialization must be applied carefully, as threshold pruning can lead to divergence. Finally, the most effective algorithm for preventing plasticity loss was the combination of L2 regularization and selective reinitialization, suggesting that combining different approaches to maintaining plasticity may be more effective than each approach in isolation.

6.4 Correlates of Plasticity Loss in Continual ImageNet

The goal of this section is to examine whether the same correlates of loss of plasticity found in fully-connected networks are also encountered in convolutional networks. Specifically, the goal is to determine whether loss of plasticity is accompanied by an accumulation of frozen units, an increase in weight magnitude, a decrease in gradient magnitude, or a decrease in the stable rank of the representation. While these correlates are not direct causes of plasticity loss, they can serve as diagnostic tools to identify when the learning system encounters pathological scenarios.

The quantities of interest were measured in the following manner. The average gradient mag-

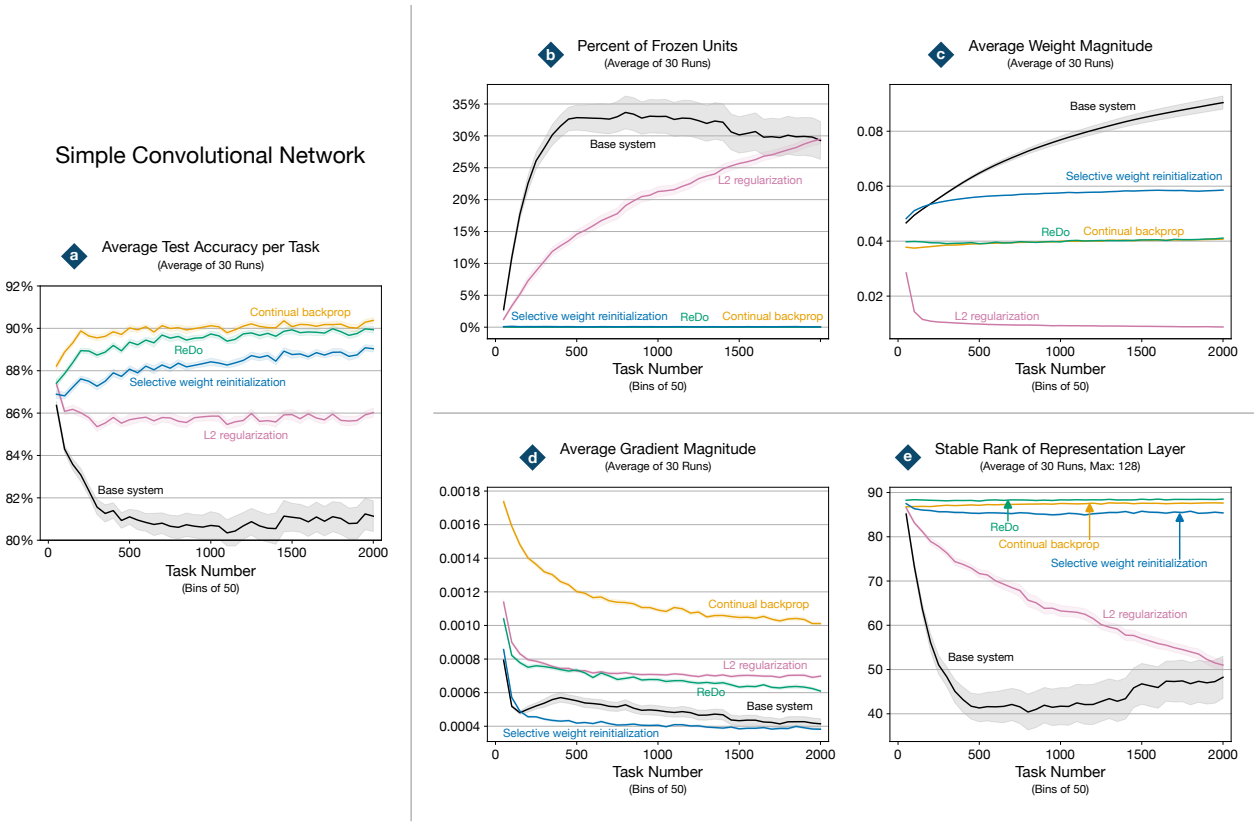

Figure 6.8: Correlates of loss of plasticity for learning systems based on the simple convolutional network architecture. **(a)** Performance of baselines and reinitialization methods on the Continual ImageNet problem. **(b)** Percent of frozen units computed on the test set at the start of each task. **(c)** Average weight magnitude computed before the start of each task. **(d)** Average gradient magnitude computed on each mini-batch of data. **(e)** Stable rank of the representation layer computed on the test set at the start of each task.

nitude was measured on every mini-batch of data as the system was learning. The average weight magnitude, the proportion of frozen units, and the stable rank of the representation were measured before each new task. For the stable rank and the proportion of frozen units, activations were computed based on the test set of the new task. Moreover, to obtain a single scalar value for activations in convolutional layers, the unit's output was defined as the average over the channel dimensions. This is the same approach used for computing the average activation of units in the utility measures of unit reinitialization algorithms.

Let us begin by examining the correlates of loss of plasticity in the simple convolutional network architecture (Figure 6.8). As the base system lost plasticity, the proportion of frozen units and the average weight magnitude increased (Figures 6.8b and 6.8c), whereas the average gradient magnitude and the stable rank decreased (Figures 6.8d and 6.8e). The same pattern is observed

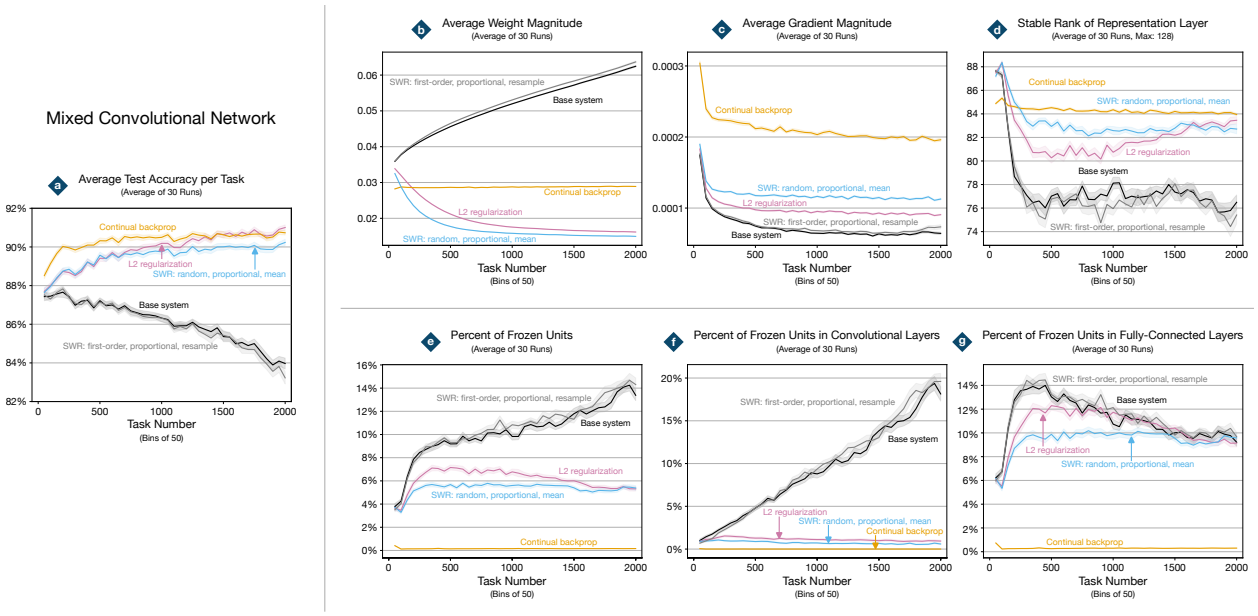

Figure 6.9: Correlates of loss of plasticity for learning systems based on the mixed convolutional network architecture. **(a)** Performance of baselines and reinitialization methods on the Continual ImageNet problem. **(b)** Average weight magnitude computed before the start of each task. **(c)** Average gradient magnitude computed on each mini-batch of data. **(d)** Stable rank of the representation layer computed on the test set at the start of each task. **(e)** Percent of frozen units in the entire network computed on the test set at the start of each task. **(f)** Percent of frozen units in the convolutional layers computed on the test set at the start of each task. **(g)** Percent of frozen units in the fully-connected layers computed on the test set at the start of each task.

in fully connected layers (see Figures 3.4 and 3.5). All the methods used to maintain plasticity prevented one or more of these pathological scenarios. Nevertheless, the algorithm with the best performance, continual backprop, prevented all of the pathological scenarios encountered in the base system.

In the mixed convolutional network architecture, the results were similar. The base system saw a deterioration in the four measurements of the correlates of loss of plasticity (Figure 6.9). Plasticity preserving methods mitigated one or more of the pathological scenarios, with the method that prevented all of them, continual backprop again, also having the highest performance. An interesting observation is that the number of frozen units showed different behaviour depending on the type of layers. In convolutional layers, there was a steady increase of frozen units in the base system (Figure 6.9f). In fully-connected layers, there was an initial increase and then a steady decrease in the number of frozen units in the base system (Figure 6.9g).

The results of this section indicate that loss of plasticity in convolutional networks is accompanied by similar qualitative changes in the learning system as in fully-connected networks. Moreover,

while not the root cause of the problem, the prevention of the correlates of loss of plasticity was also associated with better learning performance. For example, continual backprop, which had a low percent of frozen units, a high average gradient magnitude, a high stable rank, and non-increasing weight magnitude, performed the best in both convolutional network architectures.

6.5 Prior Demonstrations of Loss of Plasticity With Convolutional Networks

This section provides context about other similar demonstrations of loss of plasticity in learning systems based on similar convolutional network architectures used in this chapter.

The first clear demonstrations of plasticity loss in the literature were provided by my co-authors and me in a pre-print of our Nature paper (Dohare et al., 2023). The experiments in this chapter use the same experimental design and a similar network as those in the Nature paper. In addition, the demonstrations in this chapter strengthen the results in the Nature paper by extending the results to an additional network architecture and a different downsampled version of the ImageNet dataset. Together, these results show that a variety of convolutional neural networks lose plasticity in two different versions of the Continual ImageNet problem.

Before we demonstrated loss of plasticity, previous results showed deterioration in the performance of deep learning systems based on convolutional networks when trained on non-stationary data. The earliest demonstration was by Chaudhry et al. (2018), who showed that convolutional networks lost their ability to learn new classes in a class-incremental learning problem. These results were followed by Ash and Adams (2020), who showed that convolutional networks lost their ability to learn when trained on a small portion of the dataset before training on the full dataset. In deep reinforcement learning, researchers observed that DQN agents, which use convolutional networks similar to those used in this chapter, lost their ability to learn from new observations (Kumar et al., 2021; Nikishin et al., 2022; Lyle et al., 2022; Sokar et al., 2023). All of these results showed that the performance of learning systems based on convolutional networks deteriorated when learning from non-stationary data. However, previous work did not make the connection that these observations were different manifestations of the same underlying phenomena.

Once the loss of plasticity phenomenon caught the attention of the broader community, researchers started conducting more focused studies involving convolutional networks. In deep reinforcement learning, researchers demonstrated that Double DQN and Rainbow agents, both based on convolutional networks, also lost plasticity over time (Nikishin et al., 2023; Abbas et al., 2023). Mitigating loss of plasticity in convolutional network systems has also been a topic of extensive research. Lyle et al. (2023) and Kumar et al. (2024) both presented extensive evaluations of different methods for preventing plasticity loss in convolutional network systems in reinforcement learning

and continual supervised learning, respectively. The use of convolutional network systems and systems based on more complex architectures has become standard for comparing the capabilities of different methods for preventing plasticity loss (Lewandowski et al., 2025; Lee et al., 2024; Farias and Jozefiak, 2025).

6.6 Discussion and Conclusion

This chapter demonstrated that learning systems based on convolutional networks lose plasticity when trained on non-stationary data, which shows the phenomenon extends beyond fully-connected networks. The chapter presented a continual learning problem called the Continual ImageNet problem, which is a sequence of binary classification tasks with classes sampled from the ImageNet dataset. Learning systems based on convolutional networks showed a decrease in average test accuracy per task as they learned new tasks in the Continual ImageNet problem. If the systems were capable of learning forever, then the trend in performance should increase or remain flat. Thus, the decrease in performance is evidence that systems are slowly losing their ability to learn.

Along with the demonstrations of plasticity loss, the chapter presented baselines for maintaining plasticity in convolutional network systems. Two popular baselines in the plasticity loss literature are L2 regularization and shrink-and-perturb. When added to the simple convolutional network, both baselines mitigated plasticity loss but did not fully eliminate it. When added to the mixed convolutional network system, both baselines prevented plasticity loss and enabled performance to increase as the number of tasks increased. Thus, both of these baselines diminished the effect of plasticity loss but did not consistently remove it.

The chapter then demonstrated the effectiveness of selective reinitialization algorithms in preventing plasticity loss. The results varied widely depending on the approach and architecture. Continual backpropagation reliably prevented plasticity loss and consistently resulted in higher performance than other reinitialization approaches and the baselines. On the other hand, ReDo and selective weight reinitialization with first-order utility, threshold pruning, and resample reinitialization showed different performance depending on the architecture. When combined with the simple convolutional network system, both approaches prevented plasticity loss. However, when combined with the mixed convolutional network, ReDo performed almost as well as continual backpropagation but showed performance instability. Selective weight reinitialization initially prevented plasticity loss but eventually led to increasingly unstable performance.

The common factor between reinitialization algorithms with unstable performance was the use of threshold pruning. Using selective weight reinitialization with proportional pruning prevented learning instabilities. However, not every setting of selective weight reinitialization with proportional pruning maintained plasticity. Selective weight reinitialization with random utility and mean

reinitialization maintained plasticity in the mixed convolutional network system, albeit with lower performance than continual backprop and the two baselines. When using first-order utility and resample reinitialization, selective weight reinitialization performed the same as the base system.

Another approach to prevent learning instabilities was to use selective reinitialization combined with L2 regularization. These evaluations also allowed us to study whether reinitialization and regularization can be complementary approaches for maintaining plasticity. Combining L2 regularization with reinitialization not only prevented learning instabilities but also improved the performance of every approach. Moreover, using L2 regularization made the choice of reinitialization approach less impactful, since all approaches performed similarly.

The last studies in this chapter noted that loss of plasticity in convolutional network systems was also accompanied by the same correlates of plasticity loss observed in fully-connected network systems. Loss of plasticity in convolutional network systems was accompanied by an increase in weight magnitude and percent of frozen units, and a decrease in gradient magnitude and stable rank of the representation layer. In systems based on the mixed convolutional network, which includes both convolutional and fully-connected layers, the results were more nuanced. While the correlates remained present, different layers exhibited distinct behaviours. Specifically, the number of frozen units in the convolutional layers showed a different trend from the number of frozen units in fully-connected layers, which may indicate that plasticity loss may occur differently in different types of layers in the same architecture. Lastly, we observed that algorithms that prevented plasticity loss also prevented the correlates of plasticity loss. For example, continual backpropagation, which was the most effective at maintaining plasticity in this chapter, also prevented the increase in percent of frozen units and average weight magnitude, and the decrease in stable rank and average gradient magnitude.

The results of this chapter suggest that standard deep learning approaches may not be suitable for continual learning with convolutional network architectures. Nevertheless, these standard approaches can be augmented with selective reinitialization to enable them to learn continually from non-stationary data. While the learning problem used in this chapter may seem contrived, it illustrates scenarios often encountered in practice.

As a practitioner, imagine already having access to a neural network system that has been trained on millions of examples and has achieved a reasonable level of performance. When attempting to solve a new task related to what the network has already learned, it is tempting to simply reuse the preexisting system, as it may contain useful information. The results in the Continual ImageNet problem suggest that existing approaches may have reduced performance in such a scenario, requiring us to retrain the model from random initialization. On the other hand, selective reinitialization may enable continuous pre-training and fine-tuning of models, saving considerable time and resources. To illustrate the potential gains in performance, Figure 6.10 displays

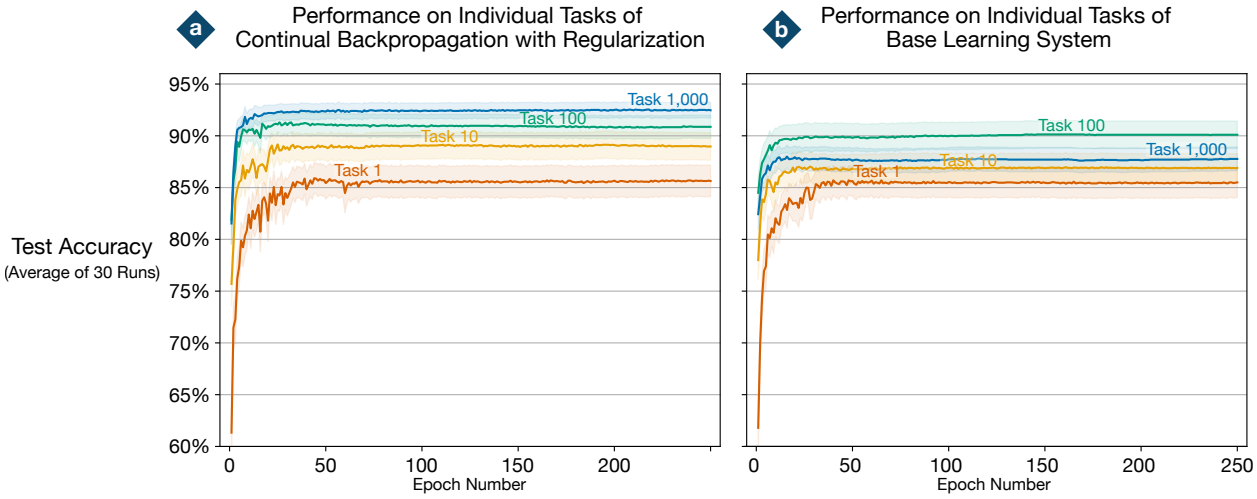

Figure 6.10: Performance on individual tasks with the mixed convolutional network. **(a)** Performance of continual backpropagation combined with L2 regularization. **(b)** Performance of base learning system. The performance of continual backpropagation with regularization keeps improving as the number of tasks increases, whereas the performance of the base learning systems deteriorates.

the performance on individual tasks of continual backpropagation combined with L2 regularization and the base learning system, both utilizing the mixed convolutional network. The performance of continual backpropagation improves as the number of tasks increases, whereas the base system's performance decreases after task 100. Additionally, during training on a new task, performance converged more quickly in later tasks than in earlier ones. Thus, the possible gains in performance are twofold. First, not having to train a network from random initialization may speed up training since performance converges faster. Second, the accumulation of knowledge across multiple tasks may enable learning systems to achieve higher performance.

The conclusions in the last two paragraphs are extrapolations from what we have observed in the Continual ImageNet. Nevertheless, the potential for such gains in performance may significantly enhance the capabilities of deep learning systems. The question is whether more complex deep learning systems also suffer plasticity loss or whether the effect is limited to the relatively simple networks we have studied so far. The next chapter investigates plasticity loss in more complex systems.

Chapter 7

Loss of Plasticity in Modern Architectures

The previous chapter extended investigations to convolutional networks, demonstrating that selective reinitialization remains effective. While the evidence establishes plasticity loss across diverse architectures, the systems evaluated thus far lack many advances used in state-of-the-art applications. This raises a crucial question: Is plasticity loss relevant in contemporary deep learning systems?

This chapter extends the demonstrations to more complex learning systems based on residual networks and vision transformers. The experiments are conducted on a class-incremental image classification problem based on the CIFAR-100 dataset. Despite incorporating regularization and techniques standard in modern applications, both systems suffer from plasticity loss. I evaluate continual backpropagation, ReDo, and selective weight reinitialization in these complex systems. All three approaches successfully maintain plasticity in residual networks, though they prove less effective in vision transformers. Nevertheless, these results establish selective reinitialization as viable for modern architectures.

This chapter includes contributions to papers published in Nature (Dohare et al., 2024) and the Conference on Lifelong Learning Agents (Hernandez-Garcia et al., 2025).

7.1 Class-Incremental Classification in CIFAR-100

The CIFAR-100 dataset is a vision dataset consisting of 100 different classes with 500 images for training and 100 images for testing for each class (Krizhevsky and Hinton, 2009). Each image is 32 pixels wide and tall with three channels corresponding to the colours red, green, and blue—the

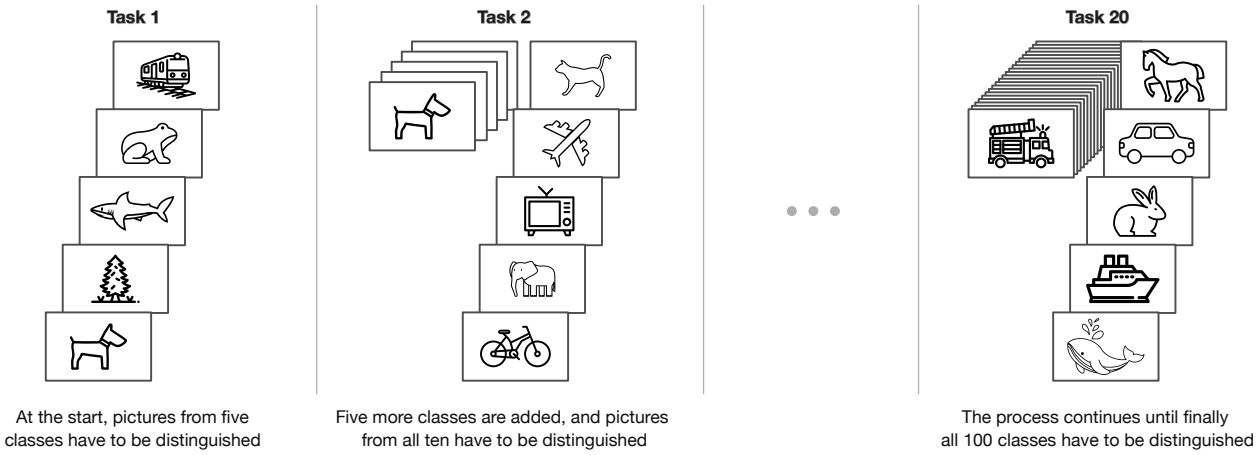

Figure 7.1: Description of the Incremental CIFAR-100 problem. Learning systems must learn to distinguish between the images in each task correctly. Tasks contain an increasing number of classes, making the classification problem increasingly difficult.

dimensions of each image are $32 \times 32 \times 3$.

The Incremental CIFAR-100 problem is a sequence of classification tasks consisting of an increasing number of classes. A learning system first learns to distinguish between images from five different classes. After several passes through the training set, also known as *epochs*, the number of classes is increased by five. The training set used to train the network is extended to include examples from all ten classes, and the learning system must now learn to distinguish between images of ten different classes. This process continues until reaching 100 classes. The Incremental CIFAR-100 problem is illustrated in Figure 7.1. For convenience, each increment of five classes is called a task. The Incremental CIFAR-100 problem consists of 20 classes in total.

The performance of a learning system is the highest accuracy measured on the test set of the current task, or simply the *top test accuracy*. This choice of performance measure is akin to how the performance of state-of-the-art systems is reported.

Note that the classification problem becomes increasingly difficult with each task since the system must learn to classify a growing number of classes. Consequently, even if a learning system is not losing plasticity, the top test accuracy would decrease as the number of tasks increases. To circumvent such an effect, plasticity is measured by comparing the performance of systems trained incrementally against the performance of a baseline system that is trained from initialization solely on the current task. For simplicity, I refer to the system trained incrementally as *incremental system*, and the system trained solely on the current task as *fresh system*. For example, when measuring plasticity during the second task, two learning systems are trained on the exact same

set of classes: the system trained incrementally since the first task—the incremental system—and a freshly initialized learning system trained only on task two—the fresh system. The measure of plasticity is the performance of the learning system trained incrementally minus the performance of the learning system trained from random initialization.

To identify plasticity loss, we must ensure that the performance of the system is not affected by any other possible detrimental phenomena. The design of the Incremental CIFAR-100 removes the effect that forgetting may have on performance. When new classes are introduced, the network is trained on examples from old and new classes, which prevents any possible forgetting that may impact performance. However, the constant retraining on old classes may result in overfitting to early classes, introducing another effect that must be accounted for. Overfitting is counteracted by using early stopping and random data transformations. Early stopping is implemented by resetting the parameters of the network to those that achieved the top test accuracy in the current task before the start of a new task. Data transformations are applied to each image as part of the data preprocessing pipeline.

Before images are fed into the network, the data is preprocessed in the following manner. First, the value of each pixel is divided by 255 to guarantee pixel values range between zero and one. Then, each channel in the image is normalized using the sample average and standard deviation computed from the entire training set. Last, to prevent overfitting, three random transformations are applied to images each time they are sampled from the training set: random horizontal flipping, random rotation, and random cropping. Random horizontal flipping is applied with a probability of 0.5. Random rotation rotates images uniformly at random between 0 and 15°. Lastly, random cropping pads the image by 4 pixels on every side and then downsamples back to the original size of the image.

The design of the Incremental CIFAR-100 prevents forgetting, and the measures taken during the training of learning systems diminish the effect of overfitting. Thus, any changes in the performance of learning systems trained incrementally relative to the fresh system should be mainly due to plasticity loss. How would the performance relative to the fresh system change depending on the plasticity of the system? If the system does not lose plasticity, the performance relative to the fresh system should stay above zero. On the other hand, if the system loses plasticity, the performance relative to the fresh system would decrease below zero. With this in mind, let us examine how networks perform when trained incrementally.

7.2 Loss of Plasticity in ResNet-18

This section examines the performance of learning systems based on residual networks in the Incremental CIFAR-100 problem. The architecture used in this work consists of 18 layers, often

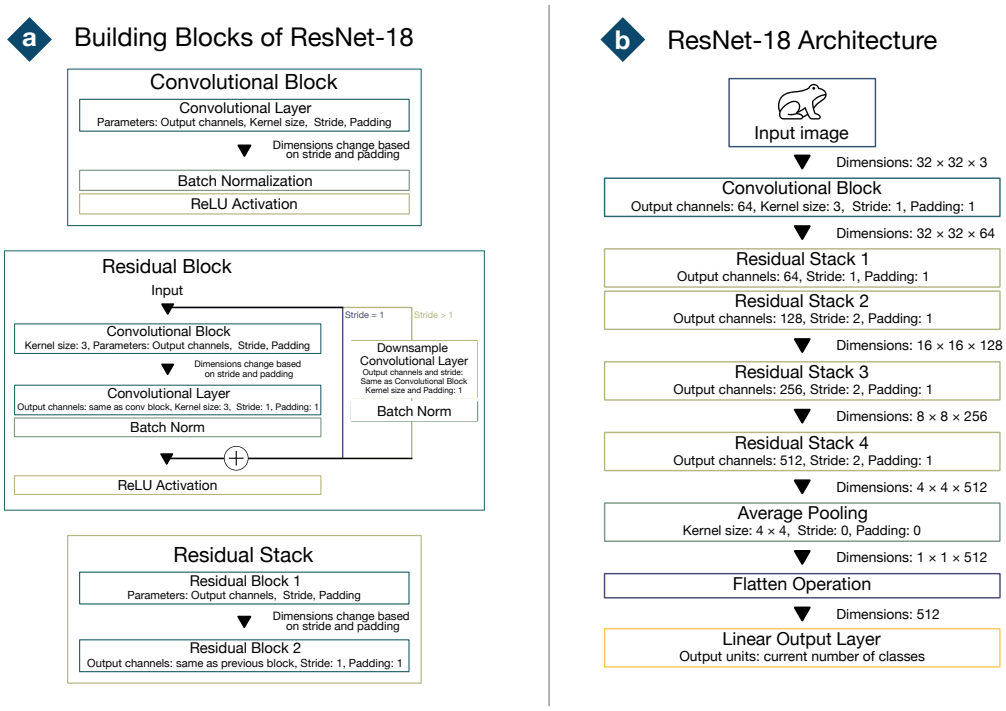

Figure 7.2: **(a)** Building blocks of the ResNet-18 architecture. **(b)** Diagram of the ResNet-18 architecture. The number of parameters in the network is $11,224,932$.

referred to as ResNet-18. The architecture consists of several building blocks illustrated in Figure 7.2a. The main innovation of the architecture is the use of skip connections, illustrated in the residual block in Figure 7.2a. The entire architecture is described in Figure 7.2b and consists of $11,224,932$ learnable parameters.

In the experiments in this chapter, ResNet-18 is trained using stochastic gradient descent (SGD) with momentum of 0.9 and L2 regularization. In each task, the network is trained for 200 epochs, resulting in a total of $4,000$ epochs of training. Additionally, the learning system employs step-size scheduling, which reduces the step-size value as the network trains. During each task, the step-size starts at a value of 0.1 and is decreased by a factor of 0.2 after 60, 120, and 160 epochs. The schedule is reset at the start of each task. The initial step-size value and the schedule were chosen according to the suggestions in Weiaicunzai (2022). The regularization factor was tuned to maximize the top accuracy in the entire dataset. Weight matrices were initialized using Kaiming Normal initialization, bias terms to zero, and the γ parameter of batch norm to one. The tuned system serves as the base system in the ResNet-18 experiments presented in this chapter.

As a baseline, the base system is augmented with shrink-and-perturb. Since the base system already includes L2 regularization, the only hyperparameter of shrink-and-perturb is the standard deviation of the noise, which was tuned to maximize the average top test accuracy across tasks.

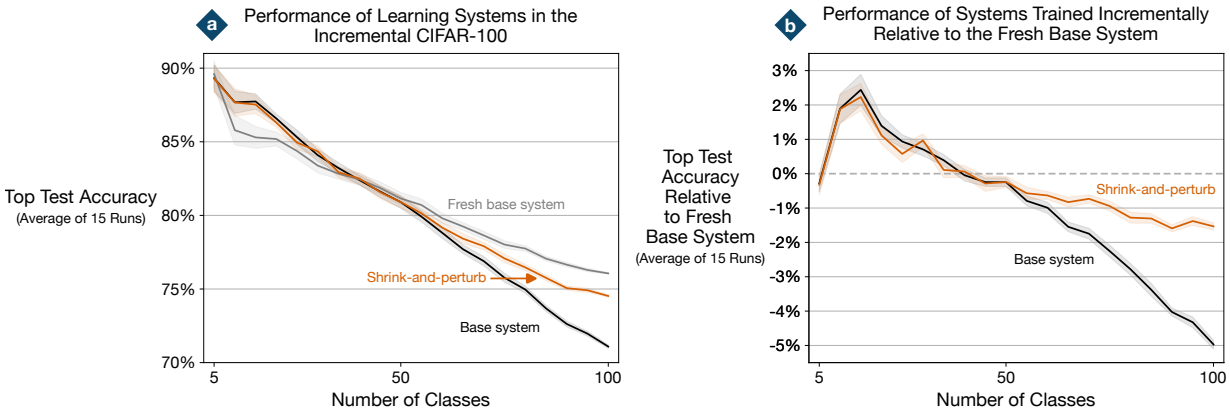

Figure 7.3: **(a)** Top test accuracy of base system and baseline. **(b)** Top test accuracy of learning systems trained incrementally and the fresh base learning system. Each line is the average of 15 runs, while the shaded region corresponds to one standard error. The zero-dash line corresponds to the performance of the fresh base system. Below the dashed line corresponds to lower performance than the fresh base system. Hyperparameters are provided in Table A.22.

The performance of the incremental base system, the fresh base system, and the shrink-and-perturb baseline is shown in Figure 7.3a. Figure 7.3b shows the performance of the incremental base and shrink-and-perturb relative to the performance of the fresh base system. The results are the average of 15 runs.

During the first tasks, the base system and shrink-and-perturb performed higher than the fresh base system (Figure 7.3). However, as new classes were introduced, the incremental learning systems failed to match the performance of the fresh base system. In other words, they did not have the same ability to learn new tasks as the fresh base system, which implies the systems had lost plasticity.

While several measures were taken to prevent overfitting, they do not rule it out. If overfitting was occurring, then classes that receive the most training would also show lower test accuracy. As a consequence, we would observe a positive correlation between the order in which a class was introduced and the test accuracy for that class on the last task. To check for any correlation, the test accuracies of each class in the final task were ranked from highest to lowest, i.e., from 0 to 99. Then, the Spearman rank correlation was measured between the ranks and the order in which each class was introduced. If the network is not overfitting, there should be no correlation between the order in which classes were introduced and the rank of the accuracies in the last task. Table 7.1 shows the correlation and a p-value corresponding to a two-sided permutation test.

The correlation results suggest that the performance gap between learning systems trained incrementally and the fresh base system was not due to overfitting. Consequently, we can conclude

Learning System	Spearman Correlation	p-value
Base system	0.0210	0.4136
Fresh base system	-0.0091	0.7106
Shrink-and-perturb	-0.0088	0.7406

Table 7.1: Correlation between the rank of the test accuracies of the 100 classes on the last task and the order in which classes were introduced during the experiment. The p-value corresponds to a two-sided permutation test where the null hypothesis is that the correlation is zero. Results are based on 15 independent runs of the experiment.

that the main effect is the loss of plasticity.

7.3 Loss of Plasticity in Vision Transformers

This section examines the performance of learning systems based on vision transformers in the Incremental CIFAR-100 problem. Vision transformers are a variant of the transformer architecture modified to process images (Dosovitskiy et al., 2021). They have become the standard approach for state-of-the-art systems dedicated to vision tasks. The main building block of the architecture is the encoder block, which consists of a multi-head self-attention layer, a fully-connected layer, and a linear layer (Figure 7.4a). The architecture used in this chapter uses a stack of eight encoder blocks (Figure 7.4b) and has a total of $14,279,140$ learnable parameters.

In the experiments in this chapter, the vision transformer is trained using SGD with momentum of 0.9, L2 regularization, and dropout. In each task, the network is trained for 100 epochs, resulting in a total of $2,000$ epochs of training. The learning system employs linear step-size scheduling, which initially increases the step-size to its maximum value over the first 30 epochs and then decreases it towards zero. Unlike the schedule used for ResNet-18, the schedule used in the vision transformer system updates the step-size after every mini-batch of data is processed. The schedule is reset at the start of each task. Tuning was performed in a similar fashion to the ResNet-18 architecture. The hyperparameters tuned were the L2 regularization factor, the maximum step-size in the scheduler, and the dropout probability. Initialization in vision transformers is a lot more intricate, so it is described in Section A.4 of Appendix A.

Finding a base system for the vision transformer system proved more challenging than for the ResNet-18 system, as several other phenomena influenced performance. First, there was the issue of overfitting. Based on five runs of the incremental vision transformer system with tuned hyperparameters, the Spearman rank correlation between the order in which classes were introduced and the rank of the accuracy for each class was 0.1269. The p-value of a two-sided permutation test was 0.0032, implying there is sufficient evidence to reject the null hypothesis that the correlation is no different from zero. This result implies that the system's predictions in the last task were

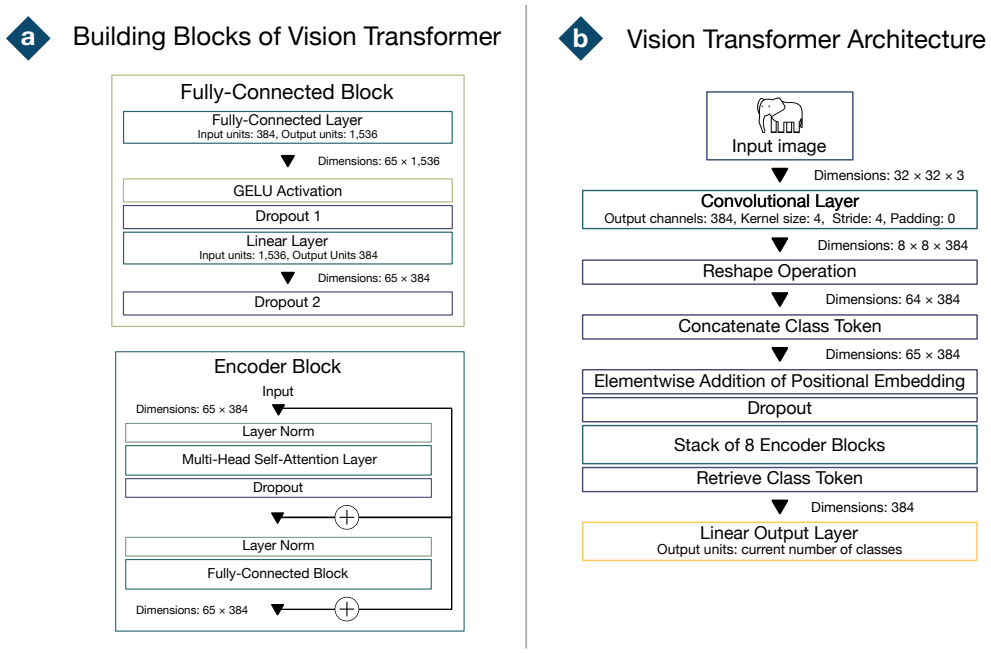

Figure 7.4: **(a)** Building blocks of the vision transformer architecture. **(b)** Diagram of the vision transformer architecture. The number of parameters in the network is $14,279,140$. The network utilized 12 attention heads per encoder block, a hidden dimension of 384, and 1,536 units in fully-connected layers.

more accurate for classes introduced late during training than for classes introduced early. Such an effect poses a problem for the experimental design, as the system should not suffer from overfitting to accurately measure plasticity loss.

To reduce the amount of overfitting, the learning system was modified to decouple the regularization factor from the optimizer's step-size. As a reminder, the SGD update with regularization is given by

$$\boldsymbol{\theta}_{t+1} \doteq \boldsymbol{\theta}_t - \alpha_t \nabla_{\boldsymbol{\theta}_t} \ell(f_{\boldsymbol{\theta}_t}(\mathbf{x}_t), \mathbf{y}_t) - \lambda \alpha_t \boldsymbol{\theta}_t, \tag{7.1}$$

where $\lambda \geq 0$ is a regularization factor and α_t is the step-size, which changes after every update due to the schedule. Decoupling involves not scaling the regularization factor by the step-size, resulting in the update rule

$$\boldsymbol{\theta}_{t+1} \doteq \boldsymbol{\theta}_t - \alpha_t \nabla_{\boldsymbol{\theta}_t} \ell(f_{\boldsymbol{\theta}_t}(\mathbf{x}_t), \mathbf{y}_t) - \lambda \boldsymbol{\theta}_t.$$

This is a more severe form of regularization, as the term $\lambda \boldsymbol{\theta}_t$ does not decrease along the step-size, which biases the iterates toward zero. Henceforth, I refer to the approach in the equation above as *decoupled regularization*, and the approach in (7.1) as *coupled regularization*.

After decoupling the regularization factor, the amount of overfitting was reduced, but another problem was encountered. The scaling parameter of layer norm operations shrank over time. As a

reminder, layer norm performs the operation:

$$y \doteq \frac{x - \overline{x}}{\sqrt{s^2 + \epsilon}} \cdot \gamma + \beta,$$

where x is an element in a vector of activations, $\overline{x}$ is the average activation in the layer, s^2 is the variance, ϵ is a small number to prevent division by zero, and γ and β are learnable parameters. In the incremental learning system with decoupled regularization, the scaling parameter, γ, shrank to extremely low values. The consequence of this phenomenon is that gradients flowing back from layer norm operations shrink by a factor of γ, which severely slows down learning. In a way, this is a form of plasticity loss that has trivial solutions for preventing it.

There are two simple approaches to preventing the scaling parameter from shrinking. The first approach is to omit regularization on the scaling parameters. Omitting regularization on vectors of parameters such as bias terms and the parameters of layer norm is common practice in deep learning (see p. 226 of Goodfellow et al., 2016). To distinguish between the two forms of applying regularization, I refer to such a regularization approach as *matrix regularization*, and the regularization approach that penalizes every parameter as *full regularization*. By excluding parameter vectors from regularization, there is no direct pressure for the scaling parameters in layer norm to decay towards zero. An alternative approach is to reparameterize layer norm in the following manner:

$$y \doteq \frac{x - \overline{x}}{\sqrt{s^2 + \epsilon}} \cdot (1 + \gamma) + \beta.$$

Such an approach does not affect the types of solutions that could be represented by layer norm. Nevertheless, it is parameterized in such a way that even if the scaling parameter, γ, is set to zero, gradients are still capable of flowing back from the operation. I call this approach *reparameterized layer norm* or *reparam LN* for short, and the approach in (7.3) as *standard layer norm* or *standard LN*.

After such a long discussion about how to apply regularization, which approach should be used as a base system? There are a total of six different candidates for the base learning system:

- with coupled and full regularization with standard layer norm, abbreviated coupled full regularization with standard LN,
- with coupled and matrix regularization with standard layer norm, coupled matrix regularization with standard LN,
- with coupled and full regularization with reparameterized layer norm, coupled full regularization with reparam LN,
- with decoupled and full regularization with standard layer norm, decoupled full regularization

with standard LN,

- with decoupled and matrix regularization with standard layer norm, decoupled matrix regularization with standard LN,
- and with decoupled and full regularization with reparameterized layer norm, decoupled full regularization with reparam LN.

While the use of colour highlighting may be a bit distracting to some readers, I found it useful in this explanation for keeping track of so many different, yet very similar, learning systems. Note that using matrix regularization with reparameterized layer norm is equivalent to using matrix regularization with standard layer norm, since either way, the scaling factor of the layer norm operation is not penalized. When tuning hyperparameters, the step-size, regularization factor, and dropout probability were tuned for the coupled full regularization with standard LN system to maximize the top test accuracy in the entire CIFAR-100 dataset. For all the other systems, only the L2 regularization factor was tuned, and they used the same step-size and dropout probability as the coupled full regularization with standard LN system.

The correlation analysis for the different systems is shown in Table 7.2. The results are based on five independent runs of the incremental systems. The systems with decoupled regularization had a lower correlation than the systems with coupled regularization. For the systems with coupled regularization, a significant correlation was observed, although we can only draw this conclusion at a 10% significance level due to multiple comparisons—the rejection p-value would be $0.1/6 \approx 0.0167$.

Table 7.2: Correlation analysis between the ranks of the accuracy in the last task and the order in which classes were introduced. The results are based on five different runs of each incremental learning system.

Learning System	**Spearman Correlation**	**P-Value**
Coupled full regularization with standard LN	0.1269	0.0032
Coupled matrix regularization with standard LN	0.1080	0.0154
Coupled full regularization with reparameterized LN	0.1163	0.0120
Decoupled full regularization with standard LN	0.0636	0.1500
Decoupled matrix regularization with standard LN	0.0679	0.1394
Decoupled full regularization with reparam LN	0.0428	0.3396

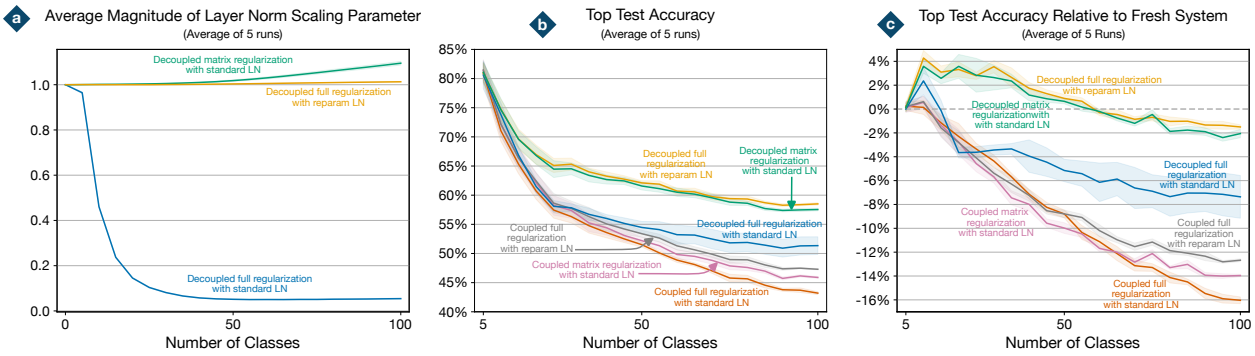

Figure 7.5: **(a)** Average magnitude of the scaling parameter in layer norm for incremental systems using decoupled regularization. The scaling factor of reparameterized layer norm is shifted up by one to account for the reparameterization. **(b)** Top test accuracy per task of the incremental learning systems. **(c)** Top test accuracy relative to fresh systems for the corresponding incremental learning system. The lines are the average of five runs. Shaded regions correspond to one standard error from the average. Hyperparameter values are provided in Table A.23.

These results suggest that decoupled regularization was successful at reducing overfitting in the learning system. If the system is not overfitting, the decrease in performance of incremental learning systems can only be attributed to loss of plasticity.

As mentioned above, the system with decoupled regularization and standard layer norm had an extreme shrinkage of the magnitude of the layer norm scaling parameter. Figure 7.5a shows the magnitude of the scaling parameter for the three systems with decoupled regularization. For the reparameterized layer norm system, the scaling factor is shifted by one to account for the reparameterization. The system with reparameterized layer norm and the system with matrix regularization both prevented this shrinkage. The average magnitude of the scaling parameter for the reparameterized layer norm system remained close to one, whereas the average magnitude for the matrix regularization system showed an increasing trend. The increasing trend in the average magnitude may constitute a problem if it were to continue, as it could amplify the magnitude of the gradients flowing back from the operation.

The proposed changes to how regularization is applied were introduced to mitigate overfitting and prevent trivial cases of plasticity loss. Ultimately, we should be most interested in how these changes affect performance and plasticity loss. Figure 7.5b shows the top test accuracy for each incremental learning system. Systems with coupled regularization performed worse than those with decoupled regularization. On the other hand, systems with reparameterized layer norm had higher performance than systems with standard layer norm, albeit only slightly higher than systems using standard layer norm and matrix regularization.

Figure 7.5c shows the performance of the incremental systems relative to the performance of

their fresh counterpart. Each line corresponds to 10 different runs, five used for the incremental system and five used for the same system trained from random initialization, the fresh system. Systems with decoupled regularization had a less steep decrease in performance as the number of tasks decreased. Since decoupled regularization systems suffered less from overfitting, we may conclude that the main culprit for this decrease in performance is the loss of plasticity. In the case of coupled regularization systems, the decrease in performance may be caused by a combination of overfitting and loss of plasticity.

Based on these results, the base system moving forward is the system with decoupled and full regularization with reparameterized layer norm. This system showed no significant overfitting, making the measurements of plasticity loss more accurate. Moreover, the system also prevented the shrinkage of the layer norm scaling parameter and achieved the best performance, while still suffering from plasticity loss, like all other incremental learning systems. The following section examines how to prevent the loss of plasticity in the ResNet-18 and vision transformer architectures using selective reinitialization.

7.4 Maintaining Plasticity in ResNet-18 and Vision Transformers With Selective Reinitialization

The previous two sections demonstrated that both the ResNet-18 learning system and the vision transformer learning system lose plasticity when learning the Incremental CIFAR-100 problem. This section shows how to augment the base systems with selective reinitialization to prevent plasticity loss. For each architecture, this section discusses how to use each selective reinitialization approach and shows how different approaches perform. Let us start with the ResNet-18 learning system.

Using unit reinitialization in ResNet-18 is performed in a manner similar to that of the convolutional networks used in the previous chapter. The channels are the basic unit in a convolutional layer. The utility of channels is computed similarly to the utility of units in fully-connected layers, except that channels are reduced to a single scalar by averaging over the width and height of the channel. Unit reinitialization is applied to every convolutional layer, except for downsample layers, which are ignored since they involve a more complex connectivity pattern. Utilities are computed based on the output of ReLU activations. When reinitializing a unit, the input weights, the corresponding entries in the batch norm layer, and the output weights corresponding to the next convolutional layer are reinitialized. The scaling parameter of batch norm is reinitialized to one, and the translation parameter is reinitialized to zero.

Using weight reinitialization in ResNet-18 is no different from using it in any other architecture, since reinitialization occurs at the level of the weights. Every vector, matrix, and tensor of

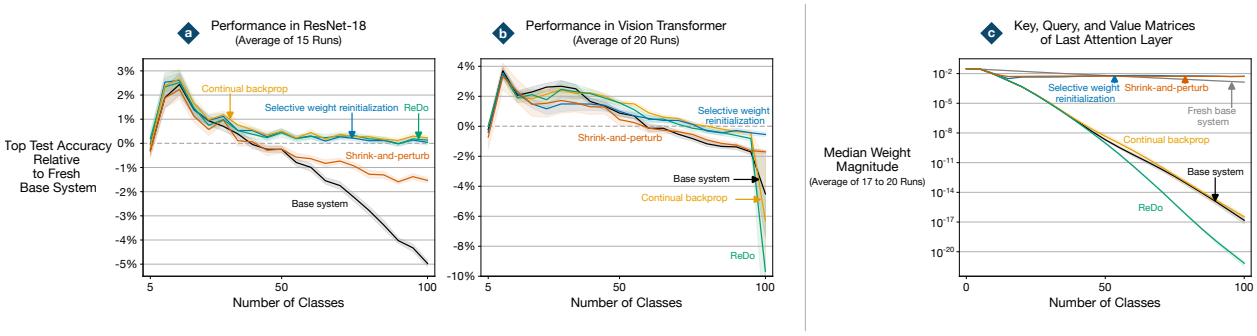

Figure 7.6: **(a)** Performance relative to the fresh base system with ResNet-18. Each line is the average of 15 runs. **(b)** Performance relative to the fresh base system with vision transformer. Each line is the average of 20 runs. **(c)** Median weight magnitude of key, query, and value matrices of the last attention layer in the vision transformer, omitting runs that diverged. ReDo's results are the average of 17 runs, whereas the results of continual backprop and the base system are the average of 19 runs. All the other lines are the average of 20 runs. Shaded regions correspond to one standard error from the average. Hyperparameters are provided in Tables A.24 and A.25.

parameters is considered when using weight reinitialization.

For unit reinitialization, both continual backprop and ReDo are used to augment the base system. Both continual backprop and ReDo use contribution utility and hemi-reinitialization. When tuning parameters, ReDo with activation utility did not improve upon the base system. This is the first setting where a noticeable difference was observed between activation and contribution utility in ReDo. For weight reinitialization, selective weight reinitialization using first-order utility with threshold pruning and resample reinitialization is used to augment the base system. All learning systems use the same step-size and L2 regularization factor as the base system, but the corresponding hyperparameters of each algorithm were tuned to maximize the average top test accuracy.

Figure 7.6a shows the performance of each selective reinitialization algorithm relative to the fresh base system. All three reinitialization algorithms maintained plasticity with no discernible differences between them.

In the vision transformer architecture, selective weight reinitialization is applied to every vector, matrix, and tensor in the architecture. Unit reinitialization, on the other hand, is only used on fully-connected layers in the fully-connected block (see Figure 7.4a). In self-attention layers, there is no clear notion of a unit, so unit reinitialization is not a feasible approach. The utility of units is computed based on the output of the GELU activation. The base system upon which reinitialization approaches are built is the system with decoupled and full regularization with reparameterized layer norm.

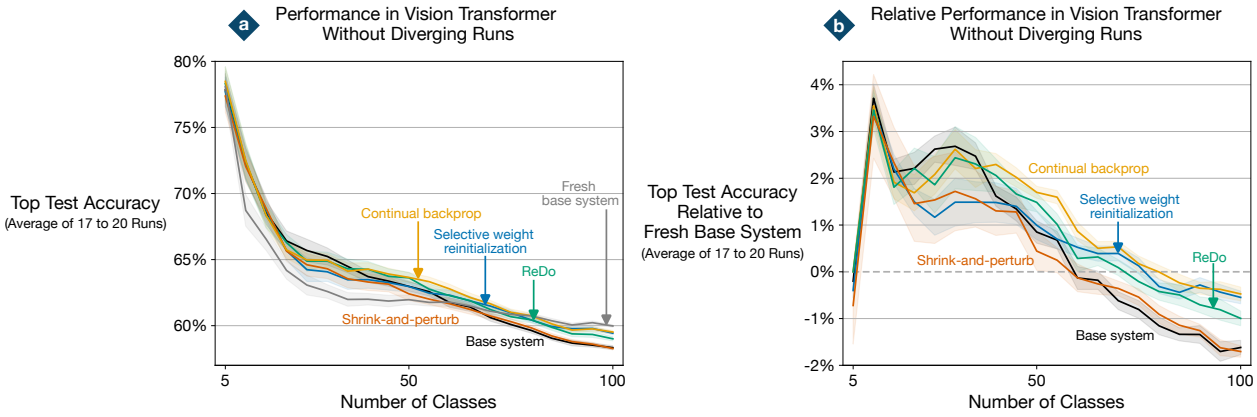

Figure 7.7: **(a)** Top test accuracy of vision transformer systems omitting runs that diverged. **(b)** Top test accuracy of vision transformer systems relative to the fresh base system without diverging runs. ReDo's results are the average of 17 runs, whereas the results continual backprop and the base system are the average of 19 runs. All the other lines are the average of 20 runs. Shaded regions correspond to one standard error from the average. All three selective reinitialization algorithms exhibited similar performance, with none entirely preventing plasticity loss. Hyperparameter values are provided in Table A.25.

Selective reinitialization was slightly less effective in the vision transformer system (Figure 7.6b). The most notable result is the unstable performance of some of the systems. In the last task, several systems diverged. Specifically, one of the runs of the base system diverged, three of the ReDo system, and one of the continual backprop system. This instability was not observed in the base system in the previous section because the analysis only included five runs, of which none diverged. The source of these learning instabilities was an extreme decline in the magnitude of the weights in the last attention layer. Figure 7.6c shows the median weight magnitude of the weights in the key, query, and value matrices in the last attention layer. The drastic decrease in weight magnitude caused numerical instabilities that eventually caused divergence. In approaches that introduced noise to the attention layers, such as shrink-and-perturb and selective weight reinitialization, the extreme decrease in weight magnitude was prevented along with divergence.

The numerical instabilities can be prevented through other approaches unrelated to plasticity loss, such as gradient or weight clipping. Let us instead examine the performance of the runs that did not diverge. Figure 7.7b shows the performance of each system relative to the fresh system. Ignoring the divergence problems, the performance of selective unit reinitialization algorithms matched that of selective weight reinitialization. This result is surprising, as unit reinitialization methods only reinitialized weights in the fully-connected block, whereas selective weight reinitialization performed reinitialization across all the layers in the network. However, while more effective than shrink-and-perturb, none of the reinitialization approaches completely prevented plasticity loss. Figure 7.7a

shows the top test accuracy of the systems for completeness.

The results in this section demonstrate that selective reinitialization is still an effective approach for preventing plasticity loss. In the ResNet-18 system, all the reinitialization approaches maintained plasticity. However, in the vision transformer system, selective reinitialization mitigated plasticity loss but did not prevent the problem. Nevertheless, selective reinitialization was still the most effective approach for mitigating plasticity loss. The results in the vision transformer system represent the frontier of the demonstrations of plasticity loss in this work and in previous literature, which is discussed later in this chapter. Such results suggest that further work is needed to guarantee plasticity in transformer architectures.

7.5 Correlates of Plasticity Loss in Incremental CIFAR-100

This section examines qualitative changes in the network of the learning systems. The goal is to find if similar pathological scenarios as the ones observed in previous chapters occur in the learning systems based on ResNet-18 and vision transformers. Specifically, we examine if the number of frozen units increases and if the stable rank of the representation decreases as systems lose plasticity.

We start our examination by taking a closer look at the ResNet-18 systems. When measuring the percent of frozen units, downsample layers were excluded from the measurement. The stable rank was computed based on the output of the flatten operation before the linear output layer (See Figure 7.2b).

Figure 7.8a shows again the performance of ResNet-18 systems relative to the fresh system. Similar to previous chapters, the decrease in performance in the base system was accompanied by an increase in the percent of frozen units (Figure 7.8b) and a decrease in the stable rank of the representation (Figure 7.8c). Just as in previous chapters, systems that maintained plasticity did not suffer from similar pathologies. Continual backprop, ReDo, and selective weight reinitialization all maintained plasticity and did not experience an increase in the percent of frozen units or a decline in the stable rank. These results suggest that, in ResNet-18 systems, the percent of frozen units and the stable rank may still serve as diagnostic tools for plasticity loss.

In vision transformer systems, the story was different once again. The stable rank of the representation was computed based on the class token. The percent of frozen units was computed based on the output of fully-connected layers.

When measuring the percent of frozen units, none of the units in any of the systems was classified as frozen. As a reminder, a GELU unit is considered frozen if $g(\mathbf{w} \cdot \mathbf{x}_i + b) < 0$ for all $\mathbf{x}_i$ in a mini-batch of data, where $\mathbf{w}$ are the input weights of the unit and b is a bias term.

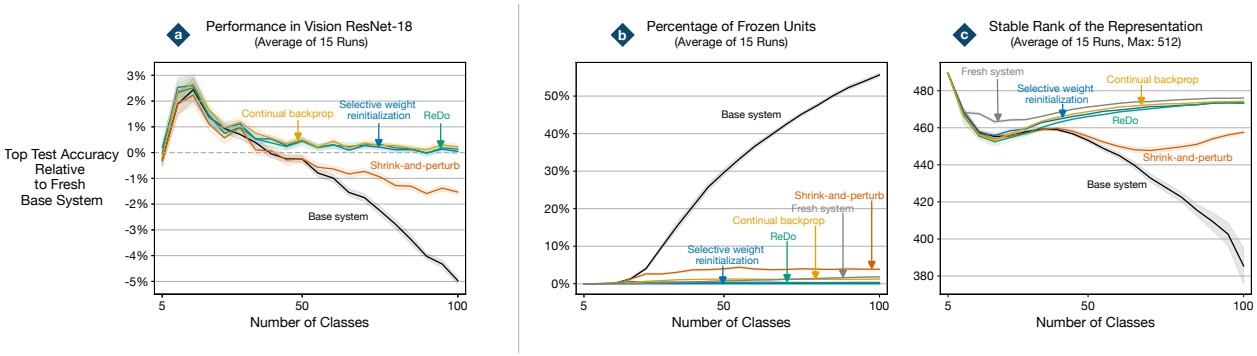

Figure 7.8: **(a)** Top test accuracy relative to fresh system of ResNet-18 systems. **(b)** Percent of frozen units in the network.. **(c)** Stable rank of the representation layer. Each line is the average of 15 runs, whereas the shaded region represents the standard error. The results resemble the results in previous chapters. Systems that lost plasticity had an increase in the percent of frozen units and a decline in stable rank.

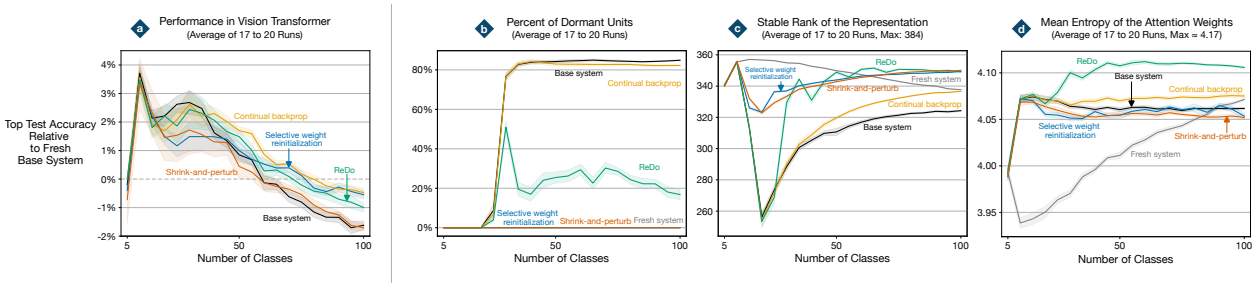

Figure 7.9: **(a)** Top test accuracy of vision transformer systems relative to the fresh base system without diverging runs. **(b)** Percent of dormant units. **(c)** Stable rank of the representation. **(d)** Entropy of attention weights. ReDo's results are the average of 17 runs, whereas the results continual backprop and the base system are the average of 19 runs. All the other lines are the average of 20 runs. Shaded regions correspond to one standard error from the average.

The initial measurements implied that none of the units consistently produced negative values, but they did not rule out the possibility of units having low activation. Let us relax the definition of frozen units to be $g(\mathbf{w} \cdot \mathbf{x}_i + b) < \epsilon$ for all $\mathbf{x}_i$, for some $\epsilon > 0$. For clarity, let us call units that meet such a criterion *dormant units*. When setting ϵ to 0.01, a clear pattern emerges for the base system. As the base system suffered from plasticity loss, the percent of dormant units in the system increased. However, the measurement was not a reliable predictor of plasticity loss. For example, continual backprop accumulated almost as many dormant units as the base system, yet it managed to mitigate plasticity loss.

The stable rank of the representation did not show any meaningful relationship with the performance of the systems. In the base system, the stable rank first declined but slowly recovered.

In systems that included continual backprop, a nearly identical pattern is observed, even though continual backprop mitigated plasticity loss. Thus, the stable rank was also not a reliable predictor of plasticity loss.

In an effort to find a predictor for plasticity loss in the vision transformer system, let us examine a new measurement. An important element of attention layers is the attention weights, which correspond to the output of the SoftMax operation in the attention layer. In a vision transformer, these weights determine which part of the image should be attended to. Parts with higher weights are given more importance and are more likely to influence the output of the network. The distribution of attention weights heavily impacts the performance of the learning system. If a few weights dominate over the rest, then the model is mostly concentrated in a few aspects of the image and ignores the rest. On the other hand, if the attention weights are uniform, then the learning system is not leveraging the attention mechanism since it does not pay attention to anything in particular. These two possible cases can be captured by measuring the entropy of the weights. Given a vector of weights $\mathbf{w} \in \mathbb{R}^n$ where $\mathbf{w}[j] \in [0, 1]$, n is the number of weights, and $\sum_{j=1}^{n} \mathbf{w}[j] = 1$, the entropy of the weights is defined as

$$\text{Entropy}(\mathbf{w}) \doteq -\sum_{j=1}^{n} \mathbf{w}[j] \log(\mathbf{w}[j]). \tag{7.2}$$

When the distribution is uniform, the entropy is equal to $\log(n)$. When the distribution is concentrated only on a few weights, the entropy approaches zero. Both scenarios are undesirable.

To see if there were any pathologies in the distribution of the attention weights, let us look at the average entropy of the attention weights across all the layers. Figure 7.9d shows the mean entropy for all the learning systems. The mean entropy implies the attention weights were close to uniform, but such was also true for the fresh system. This result has also been observed in past literature. For example, Hyeon-Woo et al. (2023) reported that attention weights in vision transformers often have high entropy. Thus, the measurements in Figure 7.9d are not uncommon for this type of architecture.

The results in this section were mixed. For ResNet-18, there were similar correlates of loss of plasticity as in previous chapters. As systems lost plasticity, their network accumulated frozen units and had a decline in stable rank. However, in vision transformer systems, no measurement reliably correlated with plasticity loss.

7.6 Prior Demonstrations With Modern Architectures in the Literature

The systems studied in this chapter represent the frontier of the types of systems used in plasticity loss research. Previous work in plasticity loss considers simpler neural network architectures, simpler problems, or both. Nevertheless, the community has slowly worked towards studying plasticity loss in more complex systems.

Several previous studies of plasticity loss using residual networks and vision transformers have done so at a smaller scale. The first study of plasticity loss using a ResNet-18 was performed by Chaudhry et al. (2018). The authors utilized a similar class-incremental setting using CIFAR-100. However, their studies placed a bigger emphasis on preventing forgetting, presenting learning systems with only five classes per task. Thus, each task consisted of an easier classification problem than the tasks in the Incremental CIFAR-100 problem in this chapter. Ash and Adams (2020) also studied how residual networks showed decreased performance in the warm-starting setting. In the warm-starting setting, a network is first trained in a subset of the dataset and subsequently on the entire dataset. The experimental setup included only a single shift in data distribution, unlike the experiments in this section, which involved several. Elsayed et al. (2024) later used the same experimental setup using a ResNet-18 system, but the learning problem still only contained a single shift in distribution. Lyle et al. (2023) presented experiments with ResNet-18 and vision transformers in a direct study of plasticity loss. Their experiments were based on a reinforcement learning problem based on the CIFAR-10 and MNIST datasets, which are smaller compared to the classification problem in this chapter. Elsayed and Mahmood (2024) used a pre-trained ResNet-50 to study plasticity loss in Continual ImageNet. However, only the fully-connected layers of the network were trained through SGD, while the rest of the network remained fixed. Farias and Jozefiak (2025) also presented studies using the Continual ImageNet problem using learning systems based on vision transformers. However, the Continual ImageNet represents a simpler problem than Incremental CIFAR-100 since each task is a binary classification problem.

Systems and problem settings that resemble the scale in this chapter have also been studied in the literature. Lee et al. (2024) study plasticity loss in learning systems based on ResNet-18 and vision transformers in non-stationary tasks based on the CIFAR-10 and CIFAR-100 datasets. The source of non-stationarity in the problem consisted of injecting normal noise into the images of the datasets with a decreasing variance, making it a smoother problem than the Incremental CIFAR-100 problem. Lewandowski et al. (2025) performed studies on plasticity loss in a class-incremental problem using the Tiny ImageNet dataset used in the previous chapter. They used a ResNet-18 and a vision transformer similar to the architectures in this section. However, they did not use a step-size schedule. To date, the demonstrations in this chapter and those by Lewandowski et al.

(2025) are the largest-scale demonstrations of plasticity loss with modern architectures.

7.7 Discussion and Conclusion

This chapter culminates the demonstrations of plasticity loss in this thesis. The results in this chapter demonstrate that even systems that use several techniques often used in state-of-the-art applications are susceptible to plasticity loss when learning from non-stationary data.

This chapter also extended the application of selective reinitialization algorithms to modern systems. In the case of selective weight reinitialization, the application was straightforward since it is readily applicable to any arbitrary network. In the case of unit reinitialization, the application of selective reinitialization was limited to only selected parts of the network. Nevertheless, both approaches proved successful at preventing plasticity loss in ResNet-18 systems. In vision transformer systems, preventing plasticity loss proved more challenging; selective reinitialization approaches only mitigated plasticity loss but did not entirely prevent it.

A significant difference in vision transformer systems to ResNet-18 systems was the step-size schedule of the optimizer. The step-size schedule was an aspect that remained unaccounted for and could have a significant impact on performance. In vision transformer systems, the step-size schedule updates the step-size after every mini-batch of data and decreases until reaching zero. The step-size schedule in ResNet-18 systems, on the other hand, only changes the step-size three times during each task and never reaches zero. The schedule in vision transformer systems could represent a problem for reinitialization algorithms because reinitialized units or weights require some amount of time to grow and mature. With a decreasing step-size, reinitialized structures receive updates of decreasing size, effectively slowing down how quickly they mature. This could result in the same structures being reinitialized repeatedly, making reinitialization less effective. An interesting avenue for future work is to account for the step-size schedule when reinitializing units or weights. This could be done by assigning a larger step-size to new weights or by using a meta-learning algorithm such as the Incremental Delta-Bar-Delta proposed by Sutton (1992) or its extension proposed by Sharifnassab et al. (2024).

Another avenue for future work is to delve deeper into how transformer systems lose plasticity. The studies about the correlates of loss of plasticity in this chapter demonstrated that plasticity loss behaves differently in transformer systems. Moreover, the results with unit reinitialization approaches demonstrated that applying reinitialization only on the fully-connected layers was enough to match the performance of selective weight reinitialization, which applied reinitialization on the entire network. Together, these results show that plasticity loss in transformers has a different behaviour than in all the architectures previously considered in this dissertation. It would be sensible to scale down the experiments to study in-depth how transformer systems lose plasticity in a

simpler and inexpensive problem that allows for multiple evaluations. In such a setting, it would also be easier to experiment and figure out how to apply selective reinitialization in transformer architectures.

While the demonstrations of plasticity loss in this chapter were insightful, they highlight that there is still much more to be done to fully understand and prevent the problem.

Chapter 8

Conclusion

This dissertation studied the problem of learning continually with deep neural networks. Continual learning could enable deep learning systems that are cheaper to update and more responsive than systems developed under the currently dominant framework. However, training deep learning systems continually has proven challenging. This work focused on the challenge of plasticity loss, or loss of the ability to learn, in deep learning systems.

This dissertation methodically built the argument that loss of plasticity is a phenomenon encountered in a wide range of deep learning systems. Demonstrations of loss of plasticity were performed in increasingly complex systems and increasingly difficult, continual supervised learning problems. In relatively simple systems, we found that techniques often employed in state-of-the-art systems, such as L2 regularization, prevented plasticity loss. However, in more complex systems, even the combination of these techniques was unable to prevent plasticity loss. In other words, techniques commonly used in deep learning systems were insufficient for robustly preventing plasticity loss in continual learning.

The solution proposed in this work for preventing plasticity loss is selective reinitialization. Selective reinitialization ensures that parameter initialization is a continuous process, rather than a one-time occurrence in the lifespan of a system. While the idea of selective reinitialization is simple, its implementation can be nuanced, requiring careful consideration.

This dissertation examined two different approaches for implementing selective reinitialization. Both approaches involve the same components, but they operate at different levels within the network: at the level of the units or at the level of the weights. The components of selective reinitialization algorithms include a utility measure that indicates the usefulness of the structure, a pruning crieterion that selects which structures to reinitialize, and a reinitialization method that assigns new values to the weights of reinitialized structures. In both approaches, the choice of each of these components was essential for ensuring the resulting system maintained plasticity.

While both selective reinitialization approaches were effective at maintaining plasticity, they offered different tradeoffs. Selective unit reinitialization proved capable of implementing reinitialization without significantly affecting the network's output. However, implementing unit reinitialization to different architectures required tailoring the technique to each specific architecture, which was not always straightforward. On the other hand, selective weight reinitialization was readily applicable to any network architecture. However, weight reinitialization often resulted in unstable performance, as the reinitialization operation frequently had a significant impact on the network's output. Nevertheless, the instability of selective weight reinitialization was ameliorated when the system included L2 regularization.

The evidence in this dissertation demonstrates that a wide range of deep learning systems are susceptible to plasticity loss; however, selective reinitialization is a robust approach for preventing the phenomenon.

While the studies in this dissertation were systematic and covered several standard techniques in deep learning, one limitation of these studies is the scale of the systems. In modern state-of-the-art applications, systems are several orders of magnitude larger than the systems studied here. As demonstrated in Chapter 3, increasing the number of parameters in a network is a viable approach for mitigating plasticity loss, albeit expensive. A remaining open question is whether scale is enough to mitigate plasticity loss in such systems.

Another avenue of future work is the refinement of selective reinitialization algorithms. There are multiple ways to improve the efficiency of the algorithms presented in Chapter 4 and 5. One straightforward approach is to propose new utility measures, pruning criteria, and reinitialization methods. Recent work on plasticity loss has already presented progress in this regard. For example, Frankle and Carbin (2019) recently proposed a different pruning criterion for unit reinitialization, while Liu et al. (2025) proposed a new utility measure similar to the first-order utility presented in Chapter 4. Moreover, there is a substantial body of work in past literature on neural network pruning that could be adapted for selective reinitialization, the details of which are presented in Chapters 4 and 5.

An orthogonal approach to improving selective reinitialization algorithms is to combine them with step size adaptation approaches. The motivation for this idea is that newly reinitialized structures are likely to require significant adjustments to their weights relative to more mature structures in the network. A way to expedite the process of adjusting the weights of new structures is by assigning a high step size to their weights. As a structure matures, it is likely to necessitate smaller adjustments to its weights, requiring a smaller step size. Thus, step-size adaptation methods such as IDBD (Sutton, 1992) could significantly improve the effectiveness of selective reinitialization algorithms.

Selective reinitialization only addresses one of the two main challenges in continual learning.

An important avenue for future work is the development of systems capable of overcoming both loss of plasticity and catastrophic forgetting simultaneously. The algorithms in this dissertation already include components that could be repurposed for preventing forgetting. Prior work has demonstrated that utility measures can be used to prevent important structures in the network from changing, resulting in less forgetting (Kirkpatrick et al., 2017; Zenke et al., 2017; Aljundi et al., 2018; Elsayed and Mahmood, 2024). In combination with prior work, the ideas in this thesis can be employed to design systems that prevent important structures from changing and selectively reinitialize unimportant ones, thereby simultaneously addressing forgetting and plasticity loss.

References

Abbas, Z., Zhao, R., Modayil, J., White, A., and Machado, M. C. (2023). Loss of plasticity in continual deep reinforcement learning. In *2nd Conference on Lifelong Learning Agents*.

Achille, A., Rovere, M., and Soatto, S. (2018). Critical learning periods in deep networks. In *6th International Conference on Learning Representations*.

Alabdulmohsin, I., Maennel, H., and Keysers, D. (2021). The impact of reinitialization on generalization in convolutional neural networks. Preprint available at `https://arxiv.org/abs/2109.00267`.

Aljundi, R., Babiloni, F., Elhoseiny, M., Rohrbach, M., and Tuytelaars, T. (2018). Memory aware synapses: Learning what (not) to forget. In *Proceedings of the European conference on computer vision (ECCV)*, pages 139–154.

Ash, J. and Adams, R. P. (2020). On warm-starting neural network training. In *Advances in Neural Information Processing Systems 33*, pages 3884–3894.

Ba, J. L., Kiros, J. R., and Hinton, G. E. (2016). Layer normalization. Preprint available at `https://arxiv.org/abs/1607.06450`.

Bahdanau, D., Cho, K., and Bengio, Y. (2014). Neural machine translation by jointly learning to align and translate. Preprint available at `https://arxiv.org/abs/1409.0473`.

Bellec, G., Kappel, D., Maass, W., and Legenstein, R. (2018). Deep rewiring: Training very sparse deep networks. In *International Conference on Learning Representations*.

Bengio, Y., Simard, P., and Frasconi, P. (1994). Learning long-term dependencies with gradient descent is difficult. *IEEE transactions on neural networks*, 5(2):157–166.

Bjorck, N., Gomes, C. P., Selman, B., and Weinberger, K. Q. (2018). Understanding batch normalization. In *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc.

Blalock, D., Gonzalez Ortiz, J. J., Frankle, J., and Guttag, J. (2020). What is the state of neural network pruning? *Proceedings of machine learning and systems*, 2:129–146.

Boyd, S. P. and Vandenberghe, L. (2004). *Convex Optimization*. Cambridge university press.

Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J. D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., et al. (2020). Language models are few-shot learners. *Advances in neural information processing systems*.

Chaudhry, A., Dokania, P. K., Ajanthan, T., and Torr, P. H. (2018). Riemannian walk for incremental learning: Understanding forgetting and intransigence. In *15th European Conference on Computer Vision*, pages 532–547.

Chung, W., Cherif, L., Precup, D., and Meger, D. (2024). Parseval regularization for continual reinforcement learning. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*.

Deng, J., Dong, W., Socher, R., Li, L.-J., Li, K., and Fei-Fei, L. (2009). Imagenet: A large-scale hierarchical image database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*, pages 248–255.

Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. (2019). Bert: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 conference of the North American chapter of the association for computational linguistics: human language technologies, volume 1 (long and short papers)*, pages 4171–4186.

Dohare, S. (2020). The interplay of search and gradient descent in semi-stationary learning problems. Master's thesis, University of Alberta.

Dohare, S., Hernandez-Garcia, J. F., Lan, Q., Rahman, P., Mahmood, A. R., and Sutton, R. S. (2024). Loss of plasticity in deep continual learning. *Nature*, 632(8026):768–774.

Dohare, S., Hernandez-Garcia, J. F., Rahman, P., Mahmood, A. R., and Sutton, R. S. (2023). Maintaining plasticity in deep continual learning. Preprint at `https://arxiv.org/abs/2306.13812`.

Dohare, S., Sutton, R. S., and Mahmood, A. R. (2021). Continual Backprop: Stochastic gradient descent with persistent randomness. Preprint at `https://arxiv.org/abs/2108.06325`.

Dong, X., Chen, S., and Pan, S. J. (2017). Learning to prune deep neural networks via layer-wise optimal brain surgeon. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, page 4860–4874, Red Hook, NY, USA. Curran Associates Inc.

D'Oro, P., Schwarzer, M., Nikishin, E., Bacon, P.-L., Bellemare, M. G., and Courville, A. (2023). Sample-efficient reinforcement learning by breaking the replay ratio barrier. In *The Eleventh International Conference on Learning Representations*.

Dosovitskiy, A., Beyer, L., Kolesnikov, A., Weissenborn, D., Zhai, X., Unterthiner, T., Dehghani, M., Minderer, M., Heigold, G., Gelly, S., Uszkoreit, J., and Houlsby, N. (2021). An image is worth 16x16 words: Transformers for image recognition at scale. In *9th International Conference on Learning Representations*.

Ellis, A. W. and Lambon Ralph, M. A. (2000). Age of acquisition effects in adult lexical processing reflect loss of plasticity in maturing systems: insights from connectionist networks. *Journal of Experimental Psychology: Learning, memory, and cognition*, 26(5):1103.

Elsayed, M., Lan, Q., Lyle, C., and Mahmood, A. R. (2024). Weight clipping for deep continual and reinforcement learning. *RLJ*, 5:2198–2217.

Elsayed, M. and Mahmood, A. R. (2024). Addressing loss of plasticity and catastrophic forgetting in continual learning. In *12th International Conference on Learning Representations*.

Evci, U., Gale, T., Menick, J., Castro, P. S., and Elsen, E. (2020). Rigging the lottery: Making all tickets winners. In *International conference on machine learning*, pages 2943–2952. PMLR.

Fahlman, S. and Lebiere, C. (1989). The cascade-correlation learning architecture. *Advances in neural information processing systems*, 2.

Fahlman, S. E. et al. (1988). *An empirical study of learning speed in back-propagation networks*. Carnegie Mellon University, Computer Science Department Pittsburgh, PA, USA.

Fang, G., Ma, X., Song, M., Bi Mi, M., and Wang, X. (2023). Depgraph: Towards any structural pruning. In *2023 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 16091–16101.

Farias, V. and Jozefiak, A. D. (2025). Self-normalized resets for plasticity in continual learning. In *The Thirteenth International Conference on Learning Representations*.

Frankle, J. and Carbin, M. (2019). The lottery ticket hypothesis: Finding sparse, trainable neural networks. In *International Conference on Learning Representations*.

French, R. M. (1999). Catastrophic forgetting in connectionist networks. *Trends in Cognitive Sciences*, 3(4):128–135.

Glorot, X. and Bengio, Y. (2010). Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, volume 9 of *Proceedings of Machine Learning Research*, pages 249–256.

Goodfellow, I., Bengio, Y., and Courville, A. (2016). *Deep Learning*. MIT Press.

Goodfellow, I., Mirza, M., Xiao, D., and Aaron Courville, Y. B. (2014). An empirical investigation of catastrophic forgeting in gradient-based neural networks. In *International Conference on Learning Representations*.

Grooten, B., Sokar, G., Dohare, S., Mocanu, E., Taylor, M. E., Pechenizkiy, M., and Mocanu, D. C. (2023). Automatic noise filtering with dynamic sparse training in deep reinforcement learning. In *Proceedings of the 2023 International Conference on Autonomous Agents and Multiagent Systems*, AAMAS '23, page 1932–1941, Richland, SC. International Foundation for Autonomous Agents and Multiagent Systems.

Han, S., Pool, J., Tran, J., and Dally, W. J. (2015). Learning both weights and connections for efficient neural networks. In *Proceedings of the 29th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'15, page 1135–1143, Cambridge, MA, USA. MIT Press.

Hassibi, B., Stork, D., and Wolff, G. (1993). Optimal brain surgeon and general network pruning. In *IEEE International Conference on Neural Networks*, pages 293–299 vol.1.

He, K., Zhang, X., Ren, S., and Sun, J. (2015). Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *IEEE International Conference on Computer Vision*, pages 1026–1034.

He, K., Zhang, X., Ren, S., and Sun, J. (2016). Deep residual learning for image recognition. In *IEEE Conference on Computer Vision and Pattern Recognition*, pages 770–778.

Hendrycks, D. and Gimpel, K. (2016). Gaussian error linear units (gelus). *arXiv preprint arXiv:1606.08415*.

Hernandez-Garcia, J. F., Dohare, S., Luo, J., and Sutton, R. S. (2025). Reinitializing weights vs units for maintaining plasticity in neural networks. In *4th Conference on Lifelong Learning Agents*.

Hinton, G. E., Srivastava, N., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. R. (2012). Improving neural networks by preventing co-adaptation of feature detectors. *arXiv preprint arXiv:1207.0580*.

Hofmann, M., Becker, M. F. P., Tetzlaff, C., and Mäder, P. (2025). Concept transfer of synaptic diversity from biological to artificial neural networks. *Nature communications*, 16(1):5112.

Holland, J. H. (1992). *Adaptation in Natural and Artificial Systems*. MIT Press.

Holland, J. H. and Reitman, J. S. (1977). Cognitive systems based on adaptive algorithms. *SIGART Bull.*, 63:49–49.

Hyeon-Woo, N., Yu-Ji, K., Heo, B., Han, D., Oh, S. J., and Oh, T.-H. (2023). Scratching visual transformer's back with uniform attention. In *Proceedings of the IEEE/CVF international conference on computer vision*, pages 5807–5818.

Igl, M., Farquhar, G., Luketina, J., Boehmer, W., and Whiteson, S. (2021). Transient non-stationarity and generalisation in deep reinforcement learning. In *International Conference on Learning Representations*.

Ioffe, S. and Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International conference on machine learning*, pages 448–456.

Javed, K. (2025). *Real-time Reinforcement Learning for Achieving Goals in Big Worlds*. PhD thesis, University of Alberta.

Kaelbling, L. P. (1993). *Learning in Embedded Systems*. MIT Press.

Kaplan, J., McCandlish, S., Henighan, T., Brown, T. B., Chess, B., Child, R., Gray, S., Radford, A., Wu, J., and Amodei, D. (2020). Scaling laws for neural language models. Preprint available at `https://arxiv.org/abs/1607.06450`.

Kasai, H., Ziv, N. E., Okazaki, H., Yagishita, S., and Toyoizumi, T. (2021). Spine dynamics in the brain, mental disorders and artificial neural networks. *Nature Reviews Neuroscience*, 22(7):407–422.

Kingma, D. P. and Ba, J. (2015). Adam: A method for stochastic optimization. In *3rd International Conference on Learning Representations*.

Kirkpatrick, J., Pascanu, R., Rabinowitz, N., Veness, J., Desjardins, G., Rusu, A. A., Milan, K., Quan, J., Ramalho, T., Grabska-Barwinska, A., et al. (2017). Overcoming catastrophic forgetting in neural networks. *Proceedings of the national academy of sciences*, 114(13):3521–3526.

Klopf, A. H. and Gose, E. (1969). An evolutionary pattern recognition network. *IEEE Transactions on Systems Science and Cybernetics*, 5(3):247–250.

Krizhevsky, A. and Hinton, G. (2009). Learning multiple layers of features from tiny images. Technical report, University of Toronto.

Kumar, A., Agarwal, R., Ghosh, D., and Levine, S. (2021). Implicit under-parameterization inhibits data-efficient deep reinforcement learning. In *International Conference on Learning Representations*.

Kumar, S., Marklund, H., and Van Roy, B. (2024). Maintaining plasticity in continual learning via regenerative regularization. In *3rd Conference on Lifelong Learning Agents*.

Le, Y. and Yang, X. (2015). Tiny imagenet visual recognition challenge. Preprint available at `https://cs231n.stanford.edu/reports/2015/pdfs/yle_project.pdf`.

Lecun, Y., Bottou, L., Bengio, Y., and Haffner, P. (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324.

LeCun, Y., Cortes, C., and Burges, C. (2010). Mnist handwritten digit database. *ATT Labs [Online]. Available: http://yann.lecun.com/exdb/mnist*, 2.

LeCun, Y., Denker, J., and Solla, S. (1989). Optimal brain damage. In Touretzky, D., editor, *Advances in Neural Information Processing Systems*, volume 2. Morgan-Kaufmann.

Lee, H., Cho, H., Kim, H., Gwak, D., Kim, J., Choo, J., Yun, S.-Y., and Yun, C. (2023). Plastic: Improving input and label plasticity for sample efficient reinforcement learning. *Advances in Neural Information Processing Systems*, 36:62270–62295.

Lee, H., Cho, H., Kim, H., Kim, D., Min, D., Choo, J., and Lyle, C. (2024). Slow and steady wins the race: Maintaining plasticity with hare and tortoise networks. In *41st International Conference on Machine Learning*.

Lee, N., Ajanthan, T., and Torr, P. (2019). SNIP: Single-shot network pruning based on connection sensitivity. In *International Conference on Learning Representations*.

Lewandowski, A., Bortkiewicz, M., Kumar, S., György, A., Schuurmans, D., Ostaszewski, M., and Machado, M. C. (2025). Learning continually by spectral regularization. In *The Thirteenth International Conference on Learning Representations*.

Lewandowski, A., Tanaka, H., Schuurmans, D., and Machado, M. C. (2024). Curvature explains loss of plasticity. Preprint available at `https://openreview.net/forum?id=SkF7NZGVr5`.

Lhoest, Q., Villanova del Moral, A., Jernite, Y., Thakur, A., von Platen, P., Patil, S., Chaumond, J., Drame, M., Plu, J., Tunstall, L., Davison, J., Šaško, M., Chhablani, G., Malik, B., Brandeis, S., Le Scao, T., Sanh, V., Xu, C., Patry, N., McMillan-Major, A., Schmid, P., Gugger, S., Delangue, C., Matussière, T., Debut, L., Bekman, S., Cistac, P., Goehringer, T., Mustar, V., Lagunas, F., Rush, A., and Wolf, T. (2021). Datasets: A community library for natural language processing. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 175–184, Online and Punta Cana, Dominican Republic. Association for Computational Linguistics.

Li, X., Xiong, H., An, H., Xu, C.-Z., and Dou, D. (2020). Rifle: Backpropagation in depth for deep transfer learning through re-initializing the fully-connected layer. In *International conference on machine learning*, pages 6010–6019. PMLR.

Liu, B. (2022). Spurious local minima are common for deep neural networks with piecewise linear activations. *IEEE Transactions on Neural Networks and Learning Systems*, 35(4):5382–5394.

Liu, J., Wu, Z., Obando-Ceron, J., Castro, P. S., Courville, A., and Pan, L. (2025). Measure gradients, not activations! enhancing neuronal activity in deep reinforcement learning. Preprint available at `https://arxiv.org/abs/2505.24061`.

Loshchilov, I. and Hutter, F. (2019). Decoupled weight decay regularization. In *International Conference on Learning Representations*.

Lu, L., Yeonjong, S., Yanhui, S., and Karniadakis, George, E. (2020). Dying relu and initialization: Theory and numerical examples. *Communications in Computational Physics*, 28(5):1671–1706.

Lyle, C., Rowland, M., and Dabney, W. (2022). Understanding and preventing capacity loss in reinforcement learning. In *10th International Conference on Learning Representations*.

Lyle, C., Zheng, Z., Khetarpal, K., Martens, J., van Hasselt, H., Pascanu, R., and Dabney, W. (2024a). Normalization and effective learning rates in reinforcement learning. Preprint available at `https://arxiv.org/abs/2407.01800`.

Lyle, C., Zheng, Z., Khetarpal, K., van Hasselt, H., Pascanu, R., Martens, J., and Dabney, W. (2024b). Disentangling the causes of plasticity loss in neural networks. In *3rd Conference on Lifelong Learning Agents*.

Lyle, C., Zheng, Z., Nikishin, E., Avila Pires, B., Pascanu, R., and Dabney, W. (2023). Understanding plasticity in neural networks. In Krause, A., Brunskill, E., Cho, K., Engelhardt, B., Sabato, S., and Scarlett, J., editors, *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pages 23190–23211. PMLR.

Maas, A. L., Hannun, A. Y., and Ng, A. Y. (2013). Rectifier nonlinearities improve neural network acoustic models. In *ICML Workshop on Deep Learning for Audio, Speech and Language Processing*.

Mahmood, A. R. and Sutton, R. S. (2013). Representation search through generate and test. In *AAAI Workshop: Learning Rich Representations from Low-Level Sensors*.

Marquez, E. S., Hare, J. S., and Niranjan, M. (2018). Deep cascade learning. *IEEE transactions on neural networks and learning systems*, 29(11):5475–5485.

McCloskey, M. and Cohen, N. J. (1989). Catastrophic interference in connectionist networks: The sequential learning problem. *Psychology of Learning and Motivation*, 24:109–165.

Mermillod, M., Bonin, P., Morisseau, T., Méot, A., and Ferrand, L. (2009). Frequency trajectory gives rise to an age-limited learning effect as a function of input-output mapping in connectionist networks. In *Proceedings of the 31th Annual Conference of the Cognitive Science Society*, pages 2322–2327. Lawrence Erlbaum Associates Mahwah, NJ.

Mocanu, D. C., Mocanu, E., Stone, P., Nguyen, P. H., Gibescu, M., and Liotta, A. (2018). Scalable training of artificial neural networks with adaptive sparse connectivity inspired by network science. *Nature communications*, 9(1):1–12.

Molchanov, P., Mallya, A., Tyree, S., Frosio, I., and Kautz, J. (2019). Importance estimation for neural network pruning. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 11264–11272.

Molchanov, P., Tyree, S., Karras, T., Aila, T., and Kautz, J. (2017). Pruning convolutional neural networks for resource efficient inference. In *International Conference on Learning Representations*.

Montavon, G., Orr, G., and Müller, K.-R. (2012). *Neural networks: tricks of the trade*, volume 7700. springer.

Mozer, M. C. and Smolensky, P. (1988). Skeletonization: A technique for trimming the fat from a network via relevance assessment. *Advances in neural information processing systems*, 1.

Munro, P. W. (1986). State-dependent factors influencing neural plasticity: A partial account of the critical period. *Parallel distributed processing: Explorations in the microstructure of cognition*, 2:471–502.

Nikishin, E., Oh, J., Ostrovski, G., Lyle, C., Pascanu, R., Dabney, W., and Barreto, A. (2023). Deep reinforcement learning with plasticity injection. In Oh, A., Naumann, T., Globerson, A., Saenko, K., Hardt, M., and Levine, S., editors, *Advances in Neural Information Processing Systems*, volume 36, pages 37142–37159. Curran Associates, Inc.

Nikishin, E., Schwarzer, M., D'Oro, P., Bacon, P.-L., and Courville, A. (2022). The primacy bias in deep reinforcement learning. In *39th International Conference on Machine Learning*, pages 16828–16847.

Park, S., Lee, J., Mo, S., and Shin, J. (2020). Lookahead: A far-sighted alternative of magnitude-based pruning. In *International Conference on Learning Representations*.

Pascanu, R., Mikolov, T., and Bengio, Y. (2013). On the difficulty of training recurrent neural networks. In *International Conference on Machine Learning*, pages 1310–1318.

Philipp, G., Song, D., and Carbonell, J. G. (2017). The exploding gradient problem demystified-definition, prevalence, impact, origin, tradeoffs, and solutions. Preprint available at `https://arxiv.org/abs/1712.05577`.

Rahman, P. (2021). Toward generate-and-test algorithms for continual feature discovery. Master's thesis, University of Alberta.

Rakitianskaia, A. and Engelbrecht, A. (2015). Measuring saturation in neural networks. In *2015 IEEE Symposium Series on Computational Intelligence*, pages 1423–1430.

Ralph, M. A. L. and Ehsan, S. (2006). Age of acquisition effects depend on the mapping between representations and the frequency of occurrence: Empirical and computational evidence. *Visual Cognition*, 13(7-8):928–948.

Ramachandran, P., Zoph, B., and Le, Q. V. (2017). Swish: a self-gated activation function. *arXiv preprint arXiv:1710.05941*.

Razin, N. and Cohen, N. (2020). Implicit regularization in deep learning may not be explainable by norms. In *Advances in Neural Information Processing Systems*, volume 33, pages 21174–21187. Curran Associates, Inc.

Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1986). Learning representations by back-propagating errors. *Nature*, 323(6088):533–536.

Rusu, A. A., Rabinowitz, N. C., Desjardins, G., Soyer, H., Kirkpatrick, J., Kavukcuoglu, K., Pascanu, R., and Hadsell, R. (2016). Progressive neural networks. *arXiv preprint arXiv:1606.04671*.

Santurkar, S., Tsipras, D., Ilyas, A., and Madry, A. (2018). How does batch normalization help optimization? *Advances in neural information processing systems*, 31.

Selfridge, O. G. (1958). Pandemonium: A paradigm for learning. In *Mechanization of Thought Processes, Proceedings of a Symposium Held at the National Physical Laboratory*, pages 511–531, Her Majesty's Stationary Office, London.

Sharifnassab, A., Salehkaleybar, S., and Sutton, R. (2024). Metaoptimize: A framework for optimizing step sizes and other meta-parameters. *arXiv preprint arXiv:2402.02342*.

Shin, Y. and Karniadakis, G. E. (2020). Trainability of relu networks and data-dependent initialization. *Journal of Machine Learning for Modeling and Computing*, 1(1):39–74.

Smith, M., Cottrell, G., and Anderson, K. (2000). The early word catches the weights. In Leen, T., Dietterich, T., and Tresp, V., editors, *Advances in Neural Information Processing Systems*, volume 13. MIT Press.

Smith, S. L., Dherin, B., Barrett, D., and De, S. (2021). On the origin of implicit regularization in stochastic gradient descent. In *International Conference on Learning Representations*.

Sokar, G., Agarwal, R., Castro, P. S., and Evci, U. (2023). The dormant neuron phenomenon in deep reinforcement learning. In *40th International Conference on Machine Learning*, pages 32145–32168.

Springer, J. M., Goyal, S., Wen, K., Kumar, T., Yue, X., Malladi, S., Neubig, G., and Raghunathan, A. (2025). Overtrained language models are harder to fine-tune. Preprint available at `https://arxiv.org/abs/2503.19206`.

Sutton, R. S. (1992). Adapting bias by gradient descent: An incremental version of delta-bar-delta. In *AAAI*, volume 92, pages 171–176. San Jose, CA.

Sutton, R. S. and Dohare, S. (2022). Maintaining plasticity in deep continual learning. Keynote presentation at Conference on Lifelong Learning Agents. Available at `https://www.youtube.com/watch?v=p_zknyfV9fY`.

Taha, A., Shrivastava, A., and Davis, L. (2021). Knowledge evolution in neural networks. In *2021 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 12838–12847.

Thomas, M. S. C. and Johnson, M. H. (2006). The computational modeling of sensitive periods. *Developmental Psychobiology*, 48(4):337–344.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I. (2017). Attention is all you need. *Advances in neural information processing systems*, 30.

Verwimp, E., Hacohen, G., and Tuytelaars, T. (2025). Same accuracy, twice as fast: continuous training surpasses retraining from scratch. Preprint available at `https://arxiv.org/abs/2502.21147`.

Waugh, S. and Adams, A. (1995). Pruning within cascade-correlation. In *Proceedings of ICNN'95 - International Conference on Neural Networks*, volume 3, pages 1206–1210 vol.3.

Weiaicunzai (2022). pytorch-cifar100. GitHub repository available at `https://github.com/weiaicunzai/pytorch-cifar100` (2022).

Yang, Y., Zhang, G., Xu, Z., and Katabi, D. (2019). Harnessing structures for value-based planning and reinforcement learning. In *7th International Conference on Learning Representations*.

Zaidi, S., Berariu, T., Kim, H., Bornschein, J., Clopath, C., Teh, Y. W., and Pascanu, R. (2023). When does re-initialization work? In Antorán, J., Blaas, A., Feng, F., Ghalebikesabi, S., Mason, I., Pradier, M. F., Rohde, D., Ruiz, F. J. R., and Schein, A., editors, *Proceedings on "I Can't Believe It's Not Better! - Understanding Deep Learning Through Empirical Falsification" at NeurIPS 2022 Workshops*, volume 187 of *Proceedings of Machine Learning Research*, pages 12–26. PMLR.

Zenke, F., Poole, B., and Ganguli, S. (2017). Continual learning through synaptic intelligence. In *International conference on machine learning*, pages 3987–3995. PMLR.

Zevin, J. D. and Seidenberg, M. S. (2002). Age of acquisition effects in word reading and other tasks. *Journal of Memory and Language*, 47(1):1–29.

Zhou, H., Vani, A., Larochelle, H., and Courville, A. (2022). Fortuitous forgetting in connectionist networks. In *International Conference on Learning Representations*.

Appendix A

Hyperparameter Tuning

The experiments in this dissertation involved an extensive parameter search to ensure fair comparisons of the algorithms. This appendix provides additional details about the specific values for each hyperparameter in each different experiment.

For each algorithm in this dissertation, hyperparameter search involved three phases. The initial phase involved a wide grid search, with a single run conducted for each parameter combination. The second phase involved a narrow grid search over a suitable range of parameter values identified during the initial phase. The narrow grid search employed multiple runs for each parameter combination to more precisely detect performance differences. The narrow search used 10 runs per parameter combination in the Permuted MNIST experiments with 10 and 100 units per layer, and five runs in the Continual ImageNet, Incremental CIFAR-100, and the Permuted MNIST experiments with 1,000 units per layers. After this second search, the parameter combination that yielded the highest performance was selected and run for an additional number of runs.

The following sections specify the parameter values tested during the narrow search for each experiment in the dissertation. The notation xey denotes the quantity $x \times 10^y$, e.g., 5e-3 corresponds to 0.005.

A.1 Permuted MNIST Experiments

Each table in this section corresponds to an experiment in the Permuted MNIST problem. The performance guiding the search was the area under the curve of the average online accuracy plot. Unless otherwise specified, the default optimizer is stochastic gradient descent (SGD), the default network size is 100 units per layer, and the default activation function is ReLU. The hyperparameter values used in the main text are in bold. The architecture in these experiments was a fully-connected

network with three hidden layers.

Different Network Sizes		
Number of Hidden Units	**Step-Size**	
10	1e-1, **5e-2**, 1e-2, 5e-3, 1e-3, 5e-4	
100	1e-1, **5e-2**, 1e-2, 5e-3, 1e-3, 5e-4	
1000	1e-1, **5e-2**, 1e-2, 5e-3, 1e-3, 5e-4	
Different Optimizers		
Optimizer	**Step-Size**	**Momentum / β_1, β_2**
SGD	1e-1, **5e-2**, 1e-2, 5e-3, 1e-3, 5e-4	-
SGD with Momentum	1e-1, 5e-2, 1e-2, **5e-3**, 1e-3, 5e-4, 1e-4, 5e-5, 1e-5, 5e-6, 1e-6, 5e-7, 1e-7	**0.9**, 0.99, 0.999
Adam	1e-2, 5e-3, 1e-3, **5e-4**, 1e-4, 5e-5	$\beta_1 \in \{\mathbf{0.9}, 0.99, 0.999\}$ $\beta_2 \in \{0.9, 0.99, \mathbf{0.999}\}$
Different Activattions		
Activation Function	**Step-Size**	
ReLU	1e-1, **5e-2**, 1e-2, 5e-3, 1e-3, 5e-4	
Sigmoid	5e-1, 1e-1, **5e-2**, 1e-2, 5e-3, 1e-3	
Tanh	5e-1, 1e-1, 5e-2, **1e-2**, 5e-3, 1e-3	
Leaky ReLU	5e-1, 1e-1, **5e-2**, 1e-2, 5e-3, 1e-3	
GELU	5e-1, 1e-1, 5e-2, **1e-2**, 5e-3, 1e-3	
SiLU	5e-1, 1e-1, **5e-2**, 1e-2, 5e-3, 1e-3	

Table A.1: Grid search for the experiments in Figures 3.2, 3.4, and 3.6. Values in bold correspond to the values used in the main text.

Moddifications to the Base System with ReLU Units		
Method	**Hyperparameter**	**Values**
Base System	Step-size, α	1e-1, **5e-2**, 1e-2, 5e-3, 1e-3, 5e-4
CReLU	Step-size, α	1e-1, **5e-2**, 1e-2, 5e-3, 1e-3, 5e-4
Dropout	Step-size, α	1e-1, **5e-2**, 1e-2, 5e-3, 1e-3
	Dropout probability, p	**0.05**, 0.1, 0.15, 0.2
Residual Connections	Step-size, α	5e-1,**1e-1**,5e-2,1e-2,5e-3,1e-3,5e-4
	Pre-activation	True, **False**
L2 Regularization	Step-size, α	5e-2
	Regularization Factor, λ	1e-3, **1e-4**, 1e-5, 1e-6, 1e-7, 1e-8
Shrink-and-Perturb	Step-size, α	5e-2
	Regularization Factor, λ	1e-4
	Noise Variance, σ^2	1e-4, 1e-5, 1e-6, **1e-7**, 1e-8
Layer Norm	Step-size, α	5e-1,**1e-1**,5e-2,1e-2,5e-3,1e-3,5e-4
	Pre-activation	True, **False**

Table A.2: Grid search for the experiments in Figures 3.5 and 3.6. In the case of layer norm and residual connections, the pre-activation parameter indicates whether the operation was apply before or after activations. All the networks had 100 units per layer and were trained using SGD.

Selective Unit Reinitialization Algorithms with Different Network Sizes		
1,000 Hidden Units		
Method	**Hyperparameter**	**Values**
Base System	Step-size, α	1e-1, **5e-2**, 1e-2, 5e-3, 1e-3, 5e-4
L2 Regularization	Regularization Factor, λ	1e-3, 1e-4, **1e-5**, 1e-6
Shrink-and-Perturb	Regularization Factor, λ	1e-5
	Noise Variance, σ^2	1e-7, **1e-8**, 1e-9, 1e-10, 1e-11
Continual Backpropagation	Maturity Threshold, M	1, 5, 10, 50, 100, 500, 1000, 5000, 10000, **50000**, 100000
	Replacement Rate, ρ	1e-3, **1e-4**, 1e-5
ReDo	Reinit Frequency, τ	2^{12}, $\mathbf{2^{13}}$, 2^{14}
	Reinit Threshold, ρ	1e-5, 1e-4, **1e-3**, 1e-2, 1e-1
100 Hidden Units		
Method	**Hyperparameter**	**Values**
Base System	Step-size, α	1e-1, **5e-2**, 1e-2, 5e-3, 1e-3, 5e-4
L2 Regularization	Regularization Factor, λ	1e-3, **1e-4**, 1e-5, 1e-6, 1e-7, 1e-8
Shrink-and-Perturb	Regularization Factor, λ	1e-4
	Noise Variance, σ^2	1e-4, 1e-5, 1e-6, **1e-7**, 1e-8
Continual Backpropagation	Maturity Threshold, M	1, 5, 10, 50, 100, **500**
	Replacement Rate, ρ	1e-1, 1e-2, 1e-3, **1e-4**, 1e-5, 1e-6
ReDo	Reinit Frequency, τ	8, **16**, 32, 64, 128, 256
	Reinit Threshold, ρ	1e-5, **1e-4**, 1e-3, 1e-2, 1e-1, 1e-0
10 Hidden Units		
Method	**Hyperparameter**	**Values**
Base System	Step-size, α	1e-1, **5e-2**, 1e-2, 5e-3, 1e-3, 5e-4
L2 Regularization	Regularization Factor, λ	1e-3, **1e-4**, 1e-5, 1e-6, 1e-7, 1e-8
Shrink-and-Perturb	Regularization Factor, λ	1e-4
	Noise Variance, σ^2	1e-4, 1e-5, 1e-6, 1e-7, 1e-8, **1e-9**, 1e-10
Continual Backpropagation	Maturity Threshold, M	1, **5**, 10, 50, 100
	Replacement Rate, ρ	1e-1, 1e-2, **1e-3**, 1e-4, 1e-5
ReDo	Reinit Frequency, τ	**8**, 16, 32, 64, 128
	Reinit Threshold, ρ	1e-5, 1e-4, 1e-3, **1e-2**, 1e-1

Table A.3: Grid search for the experiments in Figures 4.1. All the methods used the same step-size as their corresponding base system. Continual backprop used contribution utility, whereas ReDo used activation utility. Both used hemi-reinitialization.

Selective Unit Reinitialization Algorithms with Different Optimizers		
SGD with Momentum		
Method	**Hyperparameter**	**Values**
Base System	Step-size = 5e-3, Momentum = 0.9 (Table A.1)	
L2 Regularization	Regularization Factor, λ	1e-2, 1e-3, **1e-4**, 1e-5, 1e-6
Shrink-and-Perturb	Regularization Factor, λ	1e-4
	Noise Variance, σ^2	1e-4, 1e-5, 1e-6, **1e-7**, 1e-8, 1e-9
Continual Backpropagation	Maturity Threshold, M	**1**, 5, 10, 50, 100
	Replacement Rate, ρ	1e-1, 1e-2, **1e-3**, 1e-4, 1e-5
ReDo	Reinit Frequency, τ	**8**, 16, 32, 64, 128
	Reinit Threshold, ρ	1e-5, 1e-4, **1e-3**, 1e-2, 1e-1
Adam		
Method	**Hyperparameter**	**Values**
Base System	Step-size = 5e-4, $\beta_1 = 0.9$, $\beta_2 = 0.999$ (Table A.1)	
L2 Regularization	Regularization Factor, λ	1e-2, 1e-3, 1e-4, **1e-5**, 1e-6
Shrink-and-Perturb	Regularization Factor, λ	1e-5
	Noise Variance, σ^2	1e-4, 1e-5, 1e-6, 1e-7, 1e-8, **1e-9**
Continual Backpropagation	Maturity Threshold, M	**1**, 5, 10, 50, 100
	Replacement Rate, ρ	1e-1, 1e-2, **1e-3**, 1e-4, 1e-5
ReDo	Reinit Frequency, τ	8, **16**, 32, 64, 128
	Reinit Threshold, ρ	**1e-5**, 1e-4, 1e-3, 1e-2, 1e-1

Table A.4: Grid search for the experiments in Figures 4.2. All the methods used the same step-size as their corresponding base system. Continual backprop used contribution utility, whereas ReDo used activation utility. Both used hemi-reinitialization. L2 regularization and shrink-and-perturb used SGDW and AdamW with the same hyperparameters as the base system.

Selective Unit Reinitialization with Different Utility Measures		
ReLU Activation		
Method	**Hyperparameter**	**Values**
Base System	Step-size = 5e-2 (Table A.1)	
L2 Regularization	λ = 1e-4 (Table A.3)	
Shrink-and-Perturb	λ = 1e-4, σ^2 = 1e-7 (Table A.3)	
CBP: Contribution	Maturity Threshold, M	1, 5, 10, 50, 100, **500**
	Replacement Rate, ρ	1e-1, 1e-2, 1e-3, **1e-4**, 1e-5, 1e-6
CBP: Activation	Maturity Threshold, M	1, 5, **10**, 50, 100, 500
	Replacement Rate, ρ	1e-1, 1e-2, 1e-3, **1e-4**, 1e-5, 1e-6
CBP: First-Order	Maturity Threshold, M	1, **5**, 10, 50, 100, 500
	Replacement Rate, ρ	1e-1, 1e-2, 1e-3, **1e-4**, 1e-5, 1e-6
CBP: Weight Magnitude	Maturity Threshold, M	1, 5, 10, 50, 100, **500**
	Replacement Rate, ρ	1e-1, **1e-2**, 1e-3, 1e-4, 1e-5, 1e-6
CBP: Random	Maturity Threshold, M	1, 5, 10, 50, 100, **500**
	Replacement Rate, ρ	1e-1, 1e-2, 1e-3, **1e-4**, 1e-5, 1e-6
ReDo: Activation	Reinit Frequency, τ	8, **16**, 32, 64, 128, 256
	Reinit Threshold, ρ	1e-5, **1e-4**, 1e-3, 1e-2, 1e-1, 1e-0
ReDo: Contribution	Reinit Frequency, τ	8, **16**, 32, 64, 128, 256
	Reinit Threshold, ρ	1e-5, **1e-4**, 1e-3, 1e-2, 1e-1, 1e-0
ReDo: First-Order	Reinit Frequency, τ	8, 16, **32**, 64, 128, 256
	Reinit Threshold, ρ	**1e-5**, 1e-4, 1e-3, 1e-2, 1e-1, 1e-0
ReDo: Weight Magnitude	Reinit Frequency, τ	8, 16, **32**, 64, 128, 256
	Reinit Threshold, ρ	1e-5, 1e-4, **1e-3**, 1e-2, 1e-1, 1e-0
ReDo: Random	Reinit Frequency, τ	**8**, 16, 32, 64, 128, 256
	Reinit Threshold, ρ	1e-5, 1e-4, **1e-3**, 1e-2, 1e-1, 1e-0

Table A.5: Grid search for the experiments in Figure 4.4. All the methods used the same step-size as their corresponding base system. Continual backprop and ReDo used hemi-reinitialization. The networks included 100 units per layer and were trained with SGD.

Selective Unit Reinitialization Algorithms with Different Utility Measures		
Leaky ReLU Activations		
Method	**Hyperparameter**	**Values**
Base System	Step-size = 5e-2 (Table A.1)	
L2 Regularization	Regularization Factor, λ	1e-3, **1e-4**, 1e-5, 1e-6, 1e-7, 1e-8
Shrink-and-Perturb	Regularization Factor, λ	1e-4
	Noise Variance, σ^2	1e-4, 1e-5, 1e-6, **1e-7**, 1e-8
CBP: Contribution	Maturity Threshold, M	1, 5, 10, **50**, 100
	Replacement Rate, ρ	1e-1, 1e-2, **1e-3**, 1e-4, 1e-5
CBP: First-Order	Maturity Threshold, M	1, **5**, 10, 50, 100
	Replacement Rate, ρ	1e-1, 1e-2, 1e-3, **1e-4**, 1e-5
ReDo: Activation	Reinit Frequency, τ	8, 16, 32, 64, 128, **256**
	Reinit Threshold, ρ	1e-5, 1e-4, 1e-3, 1e-2, **1e-1**, 1e-0
ReDo: First-Order	Reinit Frequency, τ	8, 16, 32, **64**, 128, 256
	Reinit Threshold, ρ	1e-5, 1e-4, **1e-3**, 1e-2, 1e-1, 1e-0
Tanh Activations		
Method	**Hyperparameter**	**Values**
Base System	Step-size = 1e-2 (Table A.1)	
L2 Regularization	Regularization Factor, λ	1e-3, 1e-4, **1e-5**, 1e-6, 1e-7, 1e-8
Shrink-and-Perturb	Regularization Factor, λ	1e-5
	Noise Variance, σ^2	1e-4,1e-5,1e-6,1e-7,1e-8,1e-9,**1e-10**,1e-11,1e-12
CBP: Contribution	Maturity Threshold, M	**1**, 5, 10, 50, 100
	Replacement Rate, ρ	1e-1, 1e-2, 1e-3, **1e-4,** 1e-5
CBP: First-Order	Maturity Threshold, M	1, 5, 10, **50**, 100
	Replacement Rate, ρ	1e-1, 1e-2, 1e-3, **1e-4**, 1e-5
ReDo: Activation	Reinit Frequency, τ	8, 16, **32**, 64, 128, 256
	Reinit Threshold, ρ	1e-5, 1e-4, **1e-3**, 1e-2, 1e-1, 1e-0
ReDo: First-Order	Reinit Frequency, τ	8, 16, 32, 64, 128, **256**
	Reinit Threshold, ρ	1e-5, 1e-4, 1e-3, **1e-2**, 1e-1, 1e-0

Table A.6: Grid search for the experiments in Figures 4.5 and 4.3. All the methods used the same step-size as their corresponding base system. Continual backprop and ReDo used hemi-reinitialization. The networks included 100 units per layer and were trained with SGD.

Unit Reinitialization Method		
Method	**Hyperparameter**	**Values**
Base System	Step-size = 5e-2 (Table A.3)	
L2 Regularization	λ = 1e-4 (Table A.3)	
Shrink-and-Perturb	λ = 1e-4, σ^2 = 1e-7 (Table A.3)	
CBP: Hemi-Reinitialization	Maturity Threshold, M	1, 5, 10, 50, 100, **500**
	Replacement Rate, ρ	1e-1, 1e-2, 1e-3, **1e-4**, 1e-5, 1e-6
CBP: Full-Reinitialization	Maturity Threshold, M	1, 10, 100, 1000, **10000**, 100000
	Replacement Rate, ρ	1e-2, 1e-3, **1e-4**, 1e-5, 1e-6
ReDo: Hemi-Reinitialization	Reinit Frequency, τ	8, **16**, 32, 64, 128, 256
	Reinit Threshold, ρ	1e-5, **1e-4**, 1e-3, 1e-2, 1e-1, 1e-0
ReDo: Full-Reinitialization	Reinit Frequency, τ	$2^5, 2^6, 2^7, 2^8, 2^9, 2^{10}, 2^{11}, 2^{12}, \mathbf{2^{13}}$
	Reinit Threshold, ρ	1e-5, 1e-4, 1e-3, 1e-2, **1e-1**

Table A.7: Grid search for the experiments in Figures 4.6 and 4.7. All the methods used the same step-size as their corresponding base system. Continual backprop used contribution utility, whereas ReDo used activation utility. The networks included 100 ReLU units per layer and were trained with SGD.

Layer Norm Baselines		
Method	**Hyperparameter**	**Values**
Base System	Step-size, α	5e-1,**1e-1**,5e-2,1e-2,5e-3,1e-3,5e-4
	Pre-activation	True, **False**
L2 Regularization	Regularization Factor, λ	1e-3, 1e-4, **1e-5**, 1e-6, 1e-7, 1e-8
Shrink-and-Perturb	Regularization Factor, λ	1e-5
	Noise Variance, σ^2	1e-4, 1e-5, 1e-6, **1e-7**, 1e-8
Utility Based on Activations		
Method	**Hyperparameter**	**Values**
Continual Backpropagation	Maturity Threshold, M	**1**, 5, 10, 50, 100
	Replacement Rate, ρ	1e-1, **1e-2**, 1e-3, 1e-4, 1e-5
ReDo	Reinit Frequency, τ	**8**, 16, 32, 64, 128
	Reinit Threshold, ρ	1e-4, 1e-3, 1e-2, **1e-1**, 1e-0
Utility Based on Normalized Activations		
Method	**Hyperparameter**	**Values**
Continual Backpropagation	Maturity Threshold, M	**1**, 5, 10, 50, 100
	Replacement Rate, ρ	1e-1, 1e-2, 1e-3, 1e-4, **1e-5**, 1e-6, 1e-7
ReDo	Reinit Frequency, τ	8, 16, **32**, 64, 128
	Reinit Threshold, ρ	1e-4, 1e-3, 1e-2, **1e-1**, 1e-0

Table A.8: Grid search for the experiments in Figure 4.9. Methods used the same step-size and layer norm setting as their corresponding base system. Continual backprop used contribution utility, whereas ReDo used activation utility. The networks included 100 ReLU units per layer and were trained with SGD.

Initial Evaluations of Selective Weight Reinitialization		
Baselines		
Method	**Hyperparameters**	
Base System	Step-size = 5e-2 (Table A.3)	
L2 Regularization	λ = 1e-4 (Table A.3)	
Shrink-and-Perturb	λ = 1e-4, σ^2 = 1e-7 (Table A.3)	
Proportional Pruning with Resample Reinitialization		
Method	**Hyperparameter**	**Values**
First-Order Utility	Reinit Frequency, τ	$2^8, 2^9, 2^{10}, 2^{11}, \mathbf{2^{12}}, 2^{13}$
	Reinit Factor, ρ	0.01, 0.05, 0.1, 0.2, **0.4**, 0.8
Weight Magnitude Utility	Reinit Frequency, τ	$2^8, 2^9, 2^{10}, 2^{11}, 2^{12}, \mathbf{2^{13}}$
	Reinit Factor, ρ	0.01, 0.05, 0.1, 0.2, 0.4, **0.8**
Random Utility	Reinit Frequency, τ	$2^8, 2^9, 2^{10}, 2^{11}, \mathbf{2^{12}}, 2^{13}$
	Reinit Factor, ρ	0.01, 0.05, 0.1, **0.2**, 0.4, 0.8
Threshold Pruning with Resample Reinitialization		
Method	**Hyperparameter**	**Values**
First-Order Utility	Reinit Frequency, τ	$2^8, 2^9, 2^{10}, \mathbf{2^{11}}, 2^{12}, 2^{13}$
	Reinit Factor, ρ	1e-6, **1e-5**, 1e-4, 1e-3, 1e-2, 1e-1
Weight Magnitude Utility	Reinit Frequency, τ	$2^8, 2^9, 2^{10}, \mathbf{2^{11}}, 2^{12}, 2^{13}$
	Reinit Factor, ρ	1e-6, 1e-5, **1e-4**, 1e-3, 1e-2, 1e-1
Random Utility	Reinit Frequency, τ	$2^8, \mathbf{2^9}, 2^{10}, 2^{11}, 2^{12}, 2^{13}$
	Reinit Factor, ρ	1e-6, 1e-5, 1e-4, 1e-3, 1e-2, **1e-1**
Proportional Pruning with Mean Reinitialization		
Method	**Hyperparameter**	**Values**
First-Order Utility	Reinit Frequency, τ	$2^8, \mathbf{2^9}, 2^{10}, 2^{11}, 2^{12}, 2^{13}$
	Reinit Factor, ρ	0.01, 0.05, 0.1, **0.2**, 0.4, 0.8
Weight Magnitude Utility	Reinit Frequency, τ	$2^8, 2^9, 2^{10}, 2^{11}, 2^{12}, \mathbf{2^{13}}$
	Reinit Factor, ρ	0.01, 0.05, 0.1, 0.2, 0.4, **0.8**
Random Utility	Reinit Frequency, τ	$2^8, 2^9, \mathbf{2^{10}}, 2^{11}, 2^{12}, 2^{13}$
	Reinit Factor, ρ	0.01, **0.05**, 0.1, 0.2, 0.4, 0.8
Threshold Pruning with Mean Reinitialization		
Method	**Hyperparameter**	**Values**
First-Order Utility	Reinit Frequency, τ	$2^8, 2^9, 2^{10}, \mathbf{2^{11}}, 2^{12}, 2^{13}$
	Reinit Factor, ρ	1e-6, 1e-5, 1e-4, **1e-3**, 1e-2, 1e-1
Weight Magnitude Utility	Reinit Frequency, τ	$2^8, 2^9, \mathbf{2^{10}}, 2^{11}, 2^{12}, 2^{13}$
	Reinit Factor, ρ	1e-6, 1e-5, 1e-4, 1e-3, 1e-2, **1e-1**
Random Utility	Reinit Frequency, τ	$2^8, 2^9, \mathbf{2^{10}}, 2^{11}, 2^{12}, 2^{13}$
	Reinit Factor, ρ	1e-6, 1e-5, 1e-4, 1e-3, 1e-2, **1e-1**

Table A.9: Grid search for the experiments in Figures 5.1 and 5.2. Methods used the same step-size as the base system. The networks included 100 ReLU units per layer and were trained with SGD.

Selective Weight Reinitialization with Low Reinitialization Frequency		
Baselines		
Method	**Hyperparameters**	
Base System	Step-size = 5e-2 (Table A.3)	
L2 Regularization	λ = 1e-4 (Table A.3)	
Shrink-and-Perturb	λ = 1e-4, σ^2 = 1e-7 (Table A.3)	
Proportional Pruning with Resample Reinitialization		
Method	**Hyperparameter**	**Values**
First-Order Utility	Reinit Frequency, τ	$2^8, 2^9, \mathbf{2^{10}}$
	Reinit Factor, ρ	0.01, 0.05, 0.1, 0.2, **0.4**, 0.8
Weight Magnitude Utility	Reinit Frequency, τ	$2^8, 2^9, \mathbf{2^{10}}$
	Reinit Factor, ρ	0.01, 0.05, 0.1, 0.2, 0.4, **0.8**
Random Utility	Reinit Frequency, τ	$2^8, 2^9, \mathbf{2^{10}}$
	Reinit Factor, ρ	0.01, **0.05**, 0.1, 0.2, 0.4, 0.8
Threshold Pruning with Resample Reinitialization		
Method	**Hyperparameter**	**Values**
First-Order Utility	Reinit Frequency, τ	$2^8, 2^9, \mathbf{2^{10}}$
	Reinit Factor, ρ	**1e-6**, 1e-5, 1e-4, 1e-3, 1e-2, 1e-1
Weight Magnitude Utility	Reinit Frequency, τ	$2^8, 2^9, \mathbf{2^{10}}$
	Reinit Factor, ρ	1e-6, **1e-5**, 1e-4, 1e-3, 1e-2, 1e-1
Random Utility	Reinit Frequency, τ	$2^8, \mathbf{2^9}, 2^{10}$
	Reinit Factor, ρ	1e-6, 1e-5, 1e-4, 1e-3, 1e-2, **1e-1**
Proportional Pruning with Mean Reinitialization		
Method	**Hyperparameter**	**Values**
First-Order Utility	Reinit Frequency, τ	$2^8, \mathbf{2^9}, 2^{10}$
	Reinit Factor, ρ	0.01, 0.05, 0.1, **0.2**, 0.4, 0.8
Weight Magnitude Utility	Reinit Frequency, τ	$2^8, 2^9, \mathbf{2^{10}}$
	Reinit Factor, ρ	0.01, 0.05, 0.1, 0.2, 0.4, **0.8**
Random Utility	Reinit Frequency, τ	$2^8, 2^9, \mathbf{2^{10}}$
	Reinit Factor, ρ	0.01, **0.05**, 0.1, 0.2, 0.4, 0.8
Threshold Pruning with Mean Reinitialization		
Method	**Hyperparameter**	**Values**
First-Order Utility	Reinit Frequency, τ	$2^8, 2^9, \mathbf{2^{10}}$
	Reinit Factor, ρ	1e-6, **1e-5**, 1e-4, 1e-3, 1e-2, 1e-1
Weight Magnitude Utility	Reinit Frequency, τ	$2^8, 2^9, \mathbf{2^{10}}$
	Reinit Factor, ρ	1e-6, 1e-5, 1e-4, 1e-3, 1e-2, **1e-1**
Random Utility	Reinit Frequency, τ	$2^8, 2^9, \mathbf{2^{10}}$
	Reinit Factor, ρ	1e-6, 1e-5, 1e-4, 1e-3, 1e-2, **1e-1**

Table A.10: Grid search for the experiments in Figure 5.3. Methods used the same step-size as the base system. The networks included 100 ReLU units per layer and were trained with SGD.

Large Network (100 Units per Layer)		
Method	**Hyperparameters**	
Base System	Step-size = 5e-2 (Table A.3)	
L2 Regularization	λ = 1e-4 (Table A.3)	
Shrink-and-Perturb	λ = 1e-4, σ^2 = 1e-7 (Table A.3)	
SWR: First-Order, Threshold, Resample	$\tau = 2^{11}$, ρ = 1e-5 (Table A.9)	
SWR: First-Order, Proportional, Resample	$\tau = 2^{12}$, $\rho = 0.4$ (Table A.9)	
SWR: Random, Proportional, Resample	$\tau = 2^{12}$, $\rho = 0.2$ (Table A.9)	
SWR: Random, Proportional, Mean	$\tau = 2^{10}$, $\rho = 0.05$ (Table A.9)	
Large Network with Layer Norm		
Method	**Hyperparameter**	**Values**
Base System	Step-size = 1e-1, Pre-activation = False (Table A.8)	
L2 Regularization	λ = 1e-5 (Table A.8)	
Shrink-and-Perturb	λ = 1e-5, σ^2 = 1e-7 (Table A.8)	
SWR: First-Order, Threshold, Resample	Reinit Frequency, τ	$2^8, 2^9, 2^{10}, \mathbf{2^{11}}, 2^{12}$
	Reinit Factor, ρ	1e-6,**1e-5**,1e-4,1e-3,1e-2
SWR: First-Order, Proportional, Resample	Reinit Frequency, τ	$2^8, 2^9, 2^{10}, \mathbf{2^{11}}, 2^{12}$
	Reinit Factor, ρ	0.05,**0.1**,0.2,0.4,0.8
SWR: Random, Proportional, Resample	Reinit Frequency, τ	$2^8, 2^9, 2^{10}, 2^{11}, \mathbf{2^{12}}$
	Reinit Factor, ρ	0.01,0.03,0.05,0.1,**0.2**
SWR: Random, Proportional, Mean	Reinit Frequency, τ	$2^8, 2^9, \mathbf{2^{10}}, 2^{11}, 2^{12}$
	Reinit Factor, ρ	0.01,0.03,**0.05**,0.1,0.2

Table A.11: Grid search for the experiments in Figure 5.4. Methods used the same step-size and layer norm setting as the corresponding base system. Networks used ReLU activations and were trained with SGD.

Small Network (10 Units per Layer)		
Method	**Hyperparameter**	**Values**
Base System	Step-size = 5e-2 (Table A.3)	
L2 Regularization	λ = 1e-4 (Table A.3)	
Shrink-and-Perturb	λ = 1e-4, σ^2 = 1e-9 (Table A.3)	
SWR: First-Order, Threshold, Resample	Reinit Frequency, τ	$2^8, 2^9, 2^{10}, \mathbf{2^{11}}, 2^{12}$
	Reinit Factor, ρ	**1e-6**,1e-5,1e-4,1e-3,1e-2
SWR: First-Order, Proportional, Resample	Reinit Frequency, τ	$2^8, 2^9, 2^{10}, 2^{11}, \mathbf{2^{12}}$
	Reinit Factor, ρ	0.01,0.05,0.1,0.2,**0.4**
SWR: Random, Proportional, Resample	Reinit Frequency, τ	$2^8, \mathbf{2^9}, 2^{10}, 2^{11}, 2^{12}$
	Reinit Factor, ρ	0.01,0.03,**0.05**,0.1,0.2
SWR: Random, Proportional, Mean	Reinit Frequency, τ	$\mathbf{2^8}, 2^9, 2^{10}, 2^{11}, 2^{12}$
	Reinit Factor, ρ	**0.01**,0.03,0.05,0.1,0.2
Small Network with Layer Norm		
Method	**Hyperparameter**	**Values**
Base System	Step-size, α	5e-1,**1e-1**,5e-2,1e-2,5e-3,1e-3,5e-4
	Pre-activation	True,**False**
L2 Regularization	Regularization Factor, λ	1e-3,**1e-4**,1e-5,1e-6,1e-7,1e-8
Shrink-and-Perturb	Regularization Factor, λ	1e-4
	Noise Variance, σ^2	1e-4,1e-5,1e-6,1e-7,1e-8,1e-9,**1e-10**,1e-11
SWR: First-Order, Threshold, Resample	Reinit Frequency, τ	$2^9, 2^{10}, \mathbf{2^{11}}, 2^{12}, 2^{13}$
	Reinit Factor, ρ	1e-6,**1e-5**,1e-4,1e-3,1e-2
SWR: First-Order, Proportional, Resample	Reinit Frequency, τ	$2^8, 2^9, 2^{10}, \mathbf{2^{11}}, 2^{12}$
	Reinit Factor, ρ	0.01,0.05,0.1,**0.2**,0.4
SWR: Random, Proportional, Resample	Reinit Frequency, τ	$2^8, \mathbf{2^9}, 2^{10}, 2^{11}, 2^{12}$
	Reinit Factor, ρ	0.01,**0.03**,0.05,0.1,0.2
SWR: Random, Proportional, Mean	Reinit Frequency, τ	$2^8, \mathbf{2^9}, 2^{10}, 2^{11}, 2^{12}$
	Reinit Factor, ρ	0.01,0.03,**0.05**,0.1,0.2

Table A.12: Grid search for the experiments in Figure 5.4. Methods used the same step-size and layer norm setting as the corresponding base system. The networks used ReLU activations and were trained with SGD.

Large Network		
Method	**Hyperparameters**	
Base System	Step-size = 5e-2 (Table A.3)	
L2 Regularization	λ = 1e-4 (Table A.3)	
Shrink-and-Perturb	λ = 1e-4, σ^2 = 1e-7 (Table A.3)	
Selective Weight Reinitialization	$\tau = 2^{11}$, ρ = 1e-5 (Table A.11)	
Continual Backpropagation	M = 500, ρ = 1e-4 (Table A.5)	
ReDo	$\tau = 2^{16}$, ρ = 1e-4 (Table A.5)	
Large Network with Layer Norm		
Method	**Hyperparameter**	**Values**
Base System	Step-size = 1e-1, Pre-activation = False (Table A.8)	
L2 Regularization	λ = 1e-5 (Table A.8)	
Shrink-and-Perturb	λ = 1e-5, σ^2 = 1e-7 (Table A.8)	
Selective Weight Reinitialization	$\tau = 2^{11}$, ρ = 1e-5 (Table A.11)	
Continual Backpropagation	M = 1, ρ = 1e-2 (Table A.8)	
ReDo	τ = 8, ρ = 1e-1 (Table A.8)	
Small Network		
Method	**Hyperparameter**	**Values**
Base System	Step-size = 5e-2 (Table A.3)	
L2 Regularization	λ = 1e-4 (Table A.3)	
Shrink-and-Perturb	λ = 1e-4, σ^2 = 1e-9 (Table A.3)	
Selective Weight Reinitialization	$\tau = 2^{11}$, ρ = 1e-6 (Table A.12)	
Continual Backpropagation	M = 5, ρ = 1e-3 (Table A.3)	
ReDo	τ = 8, ρ = 1e-2 (Table A.3)	
Small Network with Layer Norm		
Method	**Hyperparameter**	**Values**
Base System	Step-size = 1e-1, Pre-activation = False (Table A.12)	
L2 Regularization	λ = 1e-4 (Table A.12)	
Shrink-and-Perturb	λ = 1e-4, σ^2 = 1e-10 (Table A.12)	
Selective Weight Reinitialization	$\tau = 2^{11}$, ρ = 1e-5 Table A.12	
Continual Backpropagation	Maturity Threshold, M	**1**,5,10,50,100
	Replacement Rate, ρ	1e-2,1e-3,1e-4,**1e-5**,1e-6
ReDo	Reinit Frequency, τ	$2^9, 2^{10}, 2^{11}, 2^{12}, 2^{13}$
	Reinit Threshold, ρ	1e-4,1e-3,1e-2,1e-1,**1e-0**

Table A.13: Grid search for the experiments in Figure 5.5. Methods used the same step-size and layer norm setting as the corresponding base system. The networks used ReLU activations and were trained with SGD. Continual backprop used contribution utility, whereas ReDo used activation utility, both based on activations. Both used hemi-reinitialization. Selective weight reinitialization used first-order utility with threshold pruning and resample reinitialization.

Adam Optimizer		
Method	**Hyperparameter**	**Values**
Base System	Step-size = 5e-4, $\beta_1 = 0.9$, $\beta_2 = 0.999$ (Table A.1)	
L2 Regularization	λ = 1e-5 (Table A.4)	
Shrink-and-Perturb	λ = 1e-5, σ^2 = 1e-9 (Table A.4)	
SWR: First-Order, Threshold, Resample	Reinit Frequency, τ	$2^8, 2^9, 2^{10}, \mathbf{2^{11}}, 2^{12}$
	Reinit Factor, ρ	**1e-6**,1e-5,1e-4,1e-3,1e-2
SWR: First-Order, Proportional, Resample	Reinit Frequency, τ	$2^8, 2^9, 2^{10}, \mathbf{2^{11}}, 2^{12}$
	Reinit Factor, ρ	0.01,0.05,0.1,0.2,**0.4**,0.8
SWR: Random, Proportional, Resample	Reinit Frequency, τ	$2^8, \mathbf{2^9}, 2^{10}, 2^{11}, 2^{12}$
	Reinit Factor, ρ	0.01,0.03,**0.05**,0.1,0.2
SWR: Random, Proportional, Mean	Reinit Frequency, τ	$2^8, 2^9, \mathbf{2^{10}}, 2^{11}, 2^{12}$
	Reinit Factor, ρ	0.01,0.03,0.05,**0.1**,0.2
SGD with Momentum		
Method	**Hyperparameter**	**Values**
Base System	Step-size = 5e-3, Momentum = 0.9 (Table A.1)	
L2 Regularization	λ = 1e-4 (Table A.4)	
Shrink-and-Perturb	λ = 1e-4, σ^2 = 1e-7 (Table A.4)	
SWR: First-Order, Threshold, Resample	Reinit Frequency, τ	$2^8, 2^9, 2^{10}, \mathbf{2^{11}}, 2^{12}$
	Reinit Factor, ρ	**1e-6**,1e-5,1e-4,1e-3,1e-2
SWR: First-Order, Proportional, Resample	Reinit Frequency, τ	$2^8, 2^9, 2^{10}, \mathbf{2^{11}}, 2^{12}$
	Reinit Factor, ρ	0.01,0.05,0.1,0.2,**0.4**,0.8
SWR: Random, Proportional, Resample	Reinit Frequency, τ	$2^8, \mathbf{2^9}, 2^{10}, 2^{11}, 2^{12}$
	Reinit Factor, ρ	0.01,0.03,**0.05**,0.1,0.2
SWR: Random, Proportional, Mean	Reinit Frequency, τ	$2^8, \mathbf{2^9}, 2^{10}, 2^{11}, 2^{12}$
	Reinit Factor, ρ	0.01,**0.03**,0.05,0.1,0.2

Table A.14: Grid search for the experiments in Figure 5.6a. Methods used the same step-size and parameters as the corresponding base system. Networks used 100 ReLU activations per layer and were trained with SGD. L2 regularization and shrink-and-perturb used AdamW and SGDW.

Adam Optimizer	
Method	**Hyperparameters**
Base System	Step-size = 5e-4, $\beta_1 = 0.9$, $\beta_2 = 0.999$ (Table A.1)
L2 Regularization	λ = 1e-5 (Table A.4)
Shrink-and-Perturb	λ = 1e-5, Noise Variance = 1e-9 (Table A.4)
Selective Weight Reinitialization	$\tau = 2^{11}$, ρ = 1e-6 (Table A.14)
Continual Backpropagation	$M = 1$, ρ = 1e-3 (Table A.4)
ReDo	$\tau = 16$, ρ = 1e-5 (Table A.4)
SGD with Momentum	
Method	**Hyperparameters**
Base System	Step-size = 5e-3, Momentum = 0.9 (Table A.1)
L2 Regularization	λ = 1e-4 (Table A.4)
Shrink-and-Perturb	λ = 1e-4, σ^2 = 1e-7 (Table A.4)
Selective Weight Reinitialization	$\tau = 2^{11}$, ρ = 1e-6 (Table A.14)
Continual Backpropagation	$M = 1$, ρ = 1e-3 (Table A.4)
ReDo	$\tau = 8$, ρ = 1e-3 (Table A.4)

Table A.15: Grid search for the experiments in Figure 5.6b. Methods used the same step-size and parameters as the corresponding base system. Networks used 100 ReLU activations per layer and were trained with SGD. Continual backprop used contribution utility, whereas ReDo used activation; both used hemi-reinitialization. Selective weight reinitialization used first-order utility with threshold pruning and resample reinitialization. L2 regularization and shrink-and-perturb used AdamW and SGDW.

Selective Weight Reinitialization with Different Activations		
Leaky ReLU Activations		
Method	**Hyperparameter**	**Values**
Base System	Step-size = 5e-2 (Table A.1)	
L2 Regularization	λ = 1e-4 (Table A.6)	
Shrink-and-Perturb	λ = 1e-4, σ^2 = 1e-7 (Table A.6)	
SWR: First-Order, Threshold, Resample	Reinit Frequency, τ	$2^9, 2^{10}, \mathbf{2^{11}}, 2^{12}, 2^{13}$
	Reinit Factor, ρ	1e-6,1e-5,1e-4,**1e-3**,1e-2
SWR: First-Order, Proportional, Resample	Reinit Frequency, τ	$2^8, 2^9, 2^{10}, \mathbf{2^{11}}, 2^{12}$
	Reinit Factor, ρ	0.01,0.05,0.1,0.2,**0.4**
SWR: Random, Proportional, Resample	Reinit Frequency, τ	$2^8, \mathbf{2^9}, 2^{10}, 2^{11}, 2^{12}$
	Reinit Factor, ρ	0.01,**0.03**,0.05,0.1,0.2
SWR: Random, Proportional, Mean	Reinit Frequency, τ	$2^8, \mathbf{2^9}, 2^{10}, 2^{11}, 2^{12}$
	Reinit Factor, ρ	0.01,**0.03**,0.05,0.1,0.2
Tanh Activations		
Method	**Hyperparameter**	**Values**
Base System	Step-size = 1e-2 (Table A.1)	
L2 Regularization	λ = 1e-5 (Table A.6)	
Shrink-and-Perturb	λ = 1e-5, σ^2 = 1e-10 (Table A.6)	
SWR: First-Order, Threshold, Resample	Reinit Frequency, τ	$2^9, \mathbf{2^{10}}, 2^{11}, 2^{12}, 2^{13}$
	Reinit Factor, ρ	**1e-6**,1e-5,1e-4,1e-3,1e-2
SWR: First-Order, Proportional, Resample	Reinit Frequency, τ	$2^8, 2^9, 2^{10}, \mathbf{2^{11}}, 2^{12}$
	Reinit Factor, ρ	0.01,0.05,**0.1**,0.2,0.4
SWR: Random, Proportional, Resample	Reinit Frequency, τ	$2^8, 2^9, 2^{10}, \mathbf{2^{11}}, 2^{12}$
	Reinit Factor, ρ	0.01,**0.03**,0.05,0.1,0.2
SWR: Random, Proportional, Mean	Reinit Frequency, τ	$2^8, \mathbf{2^9}, 2^{10}, 2^{11}, 2^{12}$
	Reinit Factor, ρ	**0.01**,0.03,0.05,0.1,0.2

Table A.16: Grid search for the experiments in Figure 5.7a. Methods used the same step-size and parameters as the corresponding base system. Networks used 100 units per layer and were trained with SGD.

Leaky ReLU Activations	
Method	**Hyperparameters**
Base System	Step-size = 5e-2 (Table A.1)
L2 Regularization	λ = 1e-4 (Table A.6)
Shrink-and-Perturb	λ = 1e-4, σ^2 = 1e-7 (Table A.6)
Selective Weight Reinitialization	$\tau = 2^9$, ρ = 0.03 (Table A.16)
Continual Backpropagation	M = 50, ρ = 1e-3 (Table A.6)
ReDo	τ = 256, ρ = 1e-1 (Table A.6)
Tanh Activations	
Method	**Hyperparameters**
Base System	Step-size = 1e-2 (Table A.1)
L2 Regularization	λ = 1e-5 (Table A.6)
Shrink-and-Perturb	λ = 1e-5, σ^2 = 1e-10 (Table A.6)
Selective Weight Reinitialization	$\tau = 2^{10}$, ρ = 1e-6 (Table A.16)
Continual Backpropagation	M = 50, ρ = 1e-4 (Table A.6)
ReDo	τ = 256, ρ = 1e-2 (Table A.6)

Table A.17: Hyperparameters for the experiments in Figure 5.7b. Methods used the same step-size and parameters as the corresponding base system. Networks used 100 units per layer and were trained with SGD. In Leaky ReLU networks, continual backprop used contribution utility, whereas ReDo used activation; both used hemi-reinitialization; selective weight reinitialization used random utility with proportional pruning and mean reinitialization. In Tanh networks, continual backprop and ReDo used first-order utility and hemi-reinitialization; selective weight reinitialization used first-order utility with threshold pruning and resample reinitialization.

Selective Weight Reinitialization in Extra Large Networks (1,000 Hidden Units)		
Method	**Hyperparameter**	**Values**
Base System	Step-size = 5e-2 (Table A.1)	
L2 Regularization	λ = 1e-5 (Table A.3)	
Shrink-and-Perturb	λ = 1e-5, σ^2 = 1e-8 (Table A.3)	
SWR: First-Order, Threshold, Resample	Reinit Frequency, τ	$2^{13}, \mathbf{2^{14}}, 2^{15}$
	Reinit Factor, ρ	1e-9,**1e-8**,1e-7,1e-6,1e-5
SWR: First-Order, Proportional, Resample	Reinit Frequency, τ	$2^{6}, 2^{7}, 2^{8}, \mathbf{2^{9}}$
	Reinit Factor, ρ	**1e-1**,5e-2,1e-2,5e-3
SWR: Random, Proportional, Resample	Reinit Frequency, τ	$2^{7}, 2^{8}, \mathbf{2^{9}}, 2^{10}$
	Reinit Factor, ρ	5e-2,**1e-2**,5e-3,1e-3
SWR: Random, Proportional, Mean	Reinit Frequency, τ	$2^{7}, 2^{8}, \mathbf{2^{9}}, 2^{10}$
	Reinit Factor, ρ	5e-2,1e-2,**5e-3**,1e-3
Reinitializing Units vs Weights in Extra Large Networks (1,000 Hidden Units)		
Method	**Hyperparameters**	
Selective Weight Reinitialization	$\tau = 2^{14}$, ρ = 1e-8	
Continual Backpropagation	M = 50000, ρ = 1e-4 (Table A.3)	
ReDo	$\tau = 2^{13}$, ρ = 1e-3 (Table A.3)	

Table A.18: Grid search for the experiments in Figure 5.8. Methods used the same step-size and parameters as the corresponding base system. Networks used ReLU activations and were trained with SGD. In the units vs weights comparisons, selective weight reinitialization uses first-order uitility with threshold pruning and resample reinitialization. Continual backprop used contribution utility, whereas ReDo used activation; both used hemi-reinitialization.

A.2 Continual ImageNet Experiments

In this experiments, the performance guiding the hyperparameter search was the area under the curve of the average test accuracy per task plot. All networks were trained using SGDW with momentum of 0.9. Note that SGDW only differs from SGD when using regularization.

Base System and Baselines in Simple Convolutional Network		
Method	**Hyperparameter**	**Values**
Base System	Step-size, α	3e-3,**1e-3**,3e-4
L2 Regularization	Regularization Factor, λ	1e-3,1e-4,**1e-5**,1e-6
Shrink-and-Perturb	Regularization Factor, λ Noise Variance, σ^2	1e-5 1e-6,1e-7,**1e-8**,1e-9,1e-10,1e-11,1e-12,1e-13,1e-14
Base System and Baselines in Mixed Convolutional Network		
Method	**Hyperparameter**	**Values**
Base System	Step-size, α	**3e-3**,1e-3,3e-4
L2 Regularization	Regularization Factor, λ	1e-5,**1e-6**,1e-7,1e-8
Shrink-and-Perturb	Regularization Factor, λ Noise Variance, σ^2	1e-6 1e-8,1e-9,1e-10,**1e-11**,1e-12,1e-13,1e-14

Table A.19: Grid search for the baselines and base systems in the simple and mixed convolutional network systems (Figures 6.2b, 6.3b, and 6.4). In the base systems, the step-size was selected to maximize initial performance; every other system optimized for the area under the curve of the average test accuracy per task plot. The baselines use the same step-size as their corresponding base system.

Selective Reinitialization in Simple Convolutional Network		
Method	**Hyperparameter**	**Values**
Base System	Step-size = 1e-3 (Table A.19)	
L2 Regularization	λ = 1e-5 (Table A.19)	
Continual Backpropagation	Maturity Threshold, M	**1**,10,100
	Replacement Rate, ρ	1e-1,**1e-2**,1e-3
ReDo	Reinit Frequency, τ	16,**32**,64
	Reinit Threshold, ρ	1e-0,**1e-1**,1e-2
Selective Weight Reinitialization	Reinit Frequency, τ	2^{9}, $\mathbf{2^{10}}$, 2^{11}
	Reinit Factor, ρ	1e-3,**1e-4**,1e-5
Selective Reinitialization in Mixed Convolutional Network		
Method	**Hyperparameter**	**Values**
Base System	Step-size = 3e-3 (Table A.19)	
L2 Regularization	λ = 1e-6 (Table A.19)	
Continual Backpropagation	Maturity Threshold, M	**1**,10,100
	Replacement Rate, ρ	**1e-2**,1e-3,1e-4
ReDo	Reinit Frequency, τ	32,**64**,128
	Reinit Threshold, ρ	1e-0,**1e-1**,1e-2
Selective Weight Reinitialization	Reinit Frequency, τ	2^{11}, $\mathbf{2^{12}}$, 2^{13}
	Reinit Factor, ρ	1e-4,1e-5,**1e-6**

Table A.20: Grid search for selective reinitialization algorithms in the simple and mixed convolutional network systems (Figure 6.5). Continual backprop used contribution utility, whereas ReDo used activation utility; both use hemi-reinitialization. Selective weight reinitialization used first-order utility with threshold pruning and resample reinitialization. Each method uses the same step-size as the base system.

Selective Weight Reinitialization with Proportional Pruning in Mixed Convolutional Network		
Method	**Hyperparameter**	**Values**
Base System	Step-size = 3e-3 (Table A.19)	
L2 Regularization	λ = 1e-6 (Table A.19)	
Continual Backpropagation	$M = 1$, ρ = 1e-2 (Table A.20)	
SWR: Random, Proportional, Mean	Reinit Frequency, τ	$2^7, 2^8, \mathbf{2^9}$
	Reinit Factor, ρ	**1e-3**,5e-4,1e-4
SWR: First-Order, Proportional, Resample	Reinit Frequency, τ	64,128,**256**
	Reinit Factor, ρ	**1e-4**,5e-5,1e-5
Selective Reinitialization Combined with L2 Regularization in Mixed Convolutional Network		
Method	**Hyperparameter**	**Values**
Base System	Step-size = 3e-3 (Table A.19)	
L2 Regularization	λ = 1e-6 (Table A.19)	
Continual Backprop + L2 Regularization	Maturity Threshold, M	**1**,10,100
	Replacement Rate, ρ	**1e-2**,1e-3,1e-4
ReDo + L2 Regularization	Reinit Frequency, τ	**16**,32,64
	Reinit Threshold, ρ	**1e-1**,1e-2,1e-3
SWR:First-Order,Thresdhold,Resample + L2 Regularization	Reinit Frequency, τ	$2^{10}, 2^{11}, \mathbf{2^{12}}$
	Reinit Factor, ρ	1e-4,1e-5,**1e-6**

Table A.21: Grid search for selective weight reinitialization with proportional pruning and for selective reinitialization algorithms combined with regularization in the mixed convolutional network systems (Figure 6.7). Continual backprop used contribution utility, whereas ReDo used activation utility; both use hemi-reinitialization. Selective weight reinitialization used first-order utility with threshold pruning and resample reinitialization when combined with L2 regularization. Each method uses the same step-size as the base system and the same regularization factor as the L2 regularization baseline.

A.3 Incremental CIFAR-100 Experiments

In this problem, the hyperparameters of base systems were tuned to maximize the top accuracy on the 100 classes of the dataset. For other methods that augment the base system, tuning maximized the top test accuracy across all 20 tasks in the problem. All networks were trained using SGD with momentum of 0.9.

Base System and Baselines in ResNet-18		
Method	**Hyperparameter**	**Values**
Base System	Initial Step-size	0.1
	L2 Regularization Factor, λ	5e-3,**5e-4**,5e-5
Shrink-and-Perturb	Standard Deviation of Noise, σ	1e-4,**1e-5**,1e-6
	L2 Regularization Factor, λ	5e-4

Table A.22: Grid search for experiments in Figure 7.3. The shrink-and-perturb baseline used the same initial step-size and schedule as the base systems.

Candidate Base Systems in Vision Transformers		
Method	**Hyperparameter**	**Values**
Coupled Full Regularization with Standard LN	Max Step-size	1e-1,**1e-2**,1e-3
	Dropout Probability, p	0.0,**0.1**,0.2
	L2 Regularization Factor, λ	1e-2,1e-3,8e-4,7e-4,**6e-4**,5e-4,3e-4,1e-4,1e-5
Coupled Matrix Regularization with Standard LN	Max Step-size	1e-2
	Dropout Probability, p	0.1
	L2 Regularization Factor, λ	1e-2,3e-3,**2e-3**,1e-3,9e-4,8e-4,1e-4,1e-5
Coupled Full Regularization with Reparameterized LN	Max Step-size	1e-2
	Dropout Probability, p	0.1
	L2 Regularization Factor, λ	1e-2,5e-3,3e-3,**2e-3**,1e-3,9e-4,8e-4,5e-4,1e-4
Decoupled Full Regularization with Standard LN	Max Step-size	1e-2
	Dropout Probability, p	0.1
	L2 Regularization Factor, λ	1e-4,1e-5,5e-6,3e-6,**2e-6**,1e-6,5e-7,1e-7
Decoupled Matrix Regularization with Standard LN	Max Step-size	1e-2
	Dropout Probability, p	0.1
	L2 Regularization Factor, λ	1e-4,3e-5,1e-5,7e-6,**6e-6**,5e-6,4e-6,3e-6,1e-6,1e-7
Decoupled Full Regularization with Reparam LN	Max Step-size	1e-2
	Dropout Probability, p	0.1
	L2 Regularization Factor, λ	1e-5,7e-6,**6e-6**,5e-6,2e-6,1e-6,5e-7,1e-7

Table A.23: Grid search for candidate base systems for the vision transformer architecture presented in Figure 7.5. Note that all candidates used the same dropout probability and step-size as the Coupled Full Regularization with Standard LN system. This was done to reduce the amount of hyperparameter tuning.

Selective Reinitialization in ResNet-18		
Method	**Hyperparameter**	**Values**
Base System	Initial step-size = 0.2, λ = 5e-4 (Table A.22)	
Shrink-and-Perturb	λ = 5e-4, σ = 1e-5 (Table A.22)	
Continual Backpropagation	Maturity Threshold, M	**1000**,10000
	Replacement Rate, ρ	1e-4,**1e-5**,1e-6
ReDo	Reinit Frequency, τ	2^9, $\mathbf{2^{10}}$, 2^{11}
	Reinit Threshold, ρ	1e-1,**5e-2**,1e-2
Selective Weight Reinitialization	Reinit Frequency, τ	2^5, $\mathbf{2^6}$, 2^7
	Reinit Factor, ρ	1e-5,**1e-6**,1e-7

Table A.24: Grid search for selective reinitialization algorithms in Figure 7.6a. Continual backprop and ReDo used contribution utility and hemi-reinitialization. Selective weight reinitialization used first-order utility with threshold pruning and resample reinitialization. Selective reininitialization methods used the same parameter settings as the base system.

Selective Reinitialization in Vision Transformers		
Method	**Hyperparameter**	**Values**
Base System	Max step-size = 1e-2, p = 0.1, λ = 6e-6 (Table A.23)	
Shrink-and-Perturb	Regularization Factor, λ	6e-6
	Noise Variance, σ^2	1e-7,5e-8,**1e-8**,5e-9,1e-9
Continual Backpropagation	Maturity Threshold, M	1000,**10000**
	Replacement Rate, ρ	1e-8,5e-8,**1e-7**,5e-7,1e-6
ReDo	Reinit Frequency, τ	2^6, $\mathbf{2^7}$, 2^8
	Reinit Threshold, ρ	0.005,0.001,**0.0005**
Selective Weight Reinitialization	Reinit Frequency, τ	2^6, $\mathbf{2^7}$, 2^8
	Reinit Factor, ρ	0.001,**0.0005**,0.0001

Table A.25: Grid search for selective reinitialization algorithms in Figures 7.6b and 7.7. Continual backprop used contribution utility, whereas ReDo used activation utility; both used hemi-reinitialization. Selective weight reinitialization used first-order utility with threshold pruning and resample reinitialization. Selective reininitialization methods used the same parameter settings as the base system. The base system was the Decoupled Full Regularization with Reparam LN in Table A.23.

A.4 Initialization of Vision Transformer

This section describes how each layer in the vision transformer architecture is initialized following the diagram in Figure 7.4b.

The initial convolutional layer, which breaks images into patches, was initialized using a truncated Normal distribution with a mean of zero. The standard deviation was $1/\sqrt{3 \cdot 4 \cdot 4}$, each number corresponding to the number of channels, kernel width, and kernel height, respectively. The truncation value was ± 2.

The class token concatenated to the input image was initialized to zero. The embedding of the positional encoding was initialized with a Normal distribution with a mean of zero and a standard deviation of 0.02. For layer norm, the weight, γ, was initialized to one, and the bias, β, was initialized to zero. For reparameterized layer norm, both γ and β were initialized to zero.

In the multi-head self-attention layers, the query, key, and value matrices were initialized using the Xavier Uniform initialization method. The output projection matrix, W_O, was initialized with a uniform distribution with bounds of $\pm\sqrt{\frac{1}{384}}$, where 384 is the embedding dimension of the attention layers. The weight matrices and bias terms of the fully-connected layer in the fully-connected block were initialized using a Uniform distribution with bounds $\pm\sqrt{\frac{1}{384}}$. The weight matrices and bias terms of the Linear layer in the fully-connected block were initialized using a Uniform distribution with bounds $\pm\sqrt{\frac{1}{1{,}536}}$, where 1,536 is the hidden dimension in the fully-connected block. Lastly, the output layer was initialized to zero.

Appendix B

Derivation of First-Order Utility

For simplicity, let us assume the network has no bias terms. As a reminder, the l-th layer has activations $\boldsymbol{h}_l \in \mathbb{R}^{d_l}$, where d_l are the number of units in the layer. The outgoing weights of the layer are stored in matrix $\mathbf{W}_l \in \mathbb{R}^{d_{l+1} \times d_l}$. The activations for layer $l+1$ are computed as $g(\mathbf{W}_l \boldsymbol{h}_l)$, where g is a nonlinear activation function.

Two quantities are needed to derive the first-order utility measure for units: the derivative of the loss with respect to each weight, $\frac{\partial \ell(\hat{\mathbf{y}}, \mathbf{y})}{\partial \mathbf{W}_l[j,i]}$, and with respect to each activation, $\frac{\partial \ell(\hat{\mathbf{y}}, \mathbf{y})}{\partial \boldsymbol{h}_l[i]}$.

Assuming we have already computed the gradients with respect to $\boldsymbol{h}_{l+1}$, $\nabla_{\boldsymbol{h}_{l+1}} \ell(\hat{\mathbf{y}}, \mathbf{y})$, the gradient of the loss with respect to $\boldsymbol{W}_l$ is given by:

$$\begin{aligned}
\nabla_{\mathbf{W}_l} \ell(\hat{\mathbf{y}}, \mathbf{y}) &= \nabla_{\boldsymbol{h}_{l+1}} \ell(\hat{\mathbf{y}}, \mathbf{y}) \nabla_{\mathbf{W}_l} g(\mathbf{W}_l \boldsymbol{h}_l) \\
&= \nabla_{\boldsymbol{h}_{l+1}} \ell(\hat{\mathbf{y}}, \mathbf{y}) \odot g'(\mathbf{W}_l \boldsymbol{h}_l) \boldsymbol{h}_l^\top \\
&= \begin{bmatrix} \vdots \\ \frac{\partial \ell(\hat{\mathbf{y}}, \mathbf{y})}{\partial \boldsymbol{h}_{l+1}[j]} \cdot g'(\mathbf{W}_l \boldsymbol{h}_l)[j] \\ \vdots \end{bmatrix} \begin{bmatrix} \dots, & \boldsymbol{h}_l[i], & \dots \end{bmatrix},
\end{aligned}$$

where g' is the derivative of the activation function and $g'(\mathbf{W}_l \boldsymbol{h}_l)[j]$ is the j-th element of $g'(\mathbf{W}_l \boldsymbol{h}_l)$. The derivative of the loss with respect to each individual weight is given by

$$\frac{\partial \ell(\hat{\mathbf{y}}, \mathbf{y})}{\partial \mathbf{W}_l[j,i]} = \frac{\partial \ell(\hat{\mathbf{y}}, \mathbf{y})}{\partial \boldsymbol{h}_{l+1}[j]} \cdot g'(\mathbf{W}_l \boldsymbol{h}_l)[j] \cdot \boldsymbol{h}_l[i]. \tag{B.1}$$

The gradient with respect to the activations, $\boldsymbol{h}_l$, is given by

$$\begin{aligned}\nabla_{\boldsymbol{h}_l}\ell(\hat{\mathbf{y}},\mathbf{y}) &= \nabla_{\boldsymbol{h}_{l+1}}\ell(\hat{\mathbf{y}},\mathbf{y})\nabla_{\boldsymbol{h}_l}g(\mathbf{W}_l\boldsymbol{h}_l)\\ &= \mathbf{W}_l^\top\left(\nabla_{\boldsymbol{h}_{l+1}}\ell(\hat{\mathbf{y}},\mathbf{y})\odot g'(\mathbf{W}_l\boldsymbol{h}_l)\right).\end{aligned}$$

Thus, the derivative with respect to each activation, $\boldsymbol{h}_l[i]$ is

$$\frac{\partial\ell(\hat{\mathbf{y}},\mathbf{y})}{\partial\boldsymbol{h}_l[i]} = \sum_{j=1}^{d_{l+1}}\mathbf{W}_l[j,i]\frac{\partial\ell(\hat{\mathbf{y}},\mathbf{y})}{\partial\boldsymbol{h}_{l+1}[j]}\cdot g'(\mathbf{W}_l\boldsymbol{h}_l)[j]. \tag{B.2}$$

Using (B.1) and (B.1), we get

$$\begin{aligned}\ell(\hat{\mathbf{y}},\mathbf{y}\mid\boldsymbol{h}_l[i]=0)-\ell(\hat{\mathbf{y}},\mathbf{y}) &\approx \ell(\hat{\mathbf{y}},\mathbf{y})-\Big(\frac{\partial\ell(\hat{\mathbf{y}},\mathbf{y})}{\partial\boldsymbol{h}_l[i]}\Big)\boldsymbol{h}_l[i]-\ell(\hat{\mathbf{y}},\mathbf{y})\\ &= -\frac{\partial\ell(\hat{\mathbf{y}},\mathbf{y})}{\partial\boldsymbol{h}_l[i]}\boldsymbol{h}_l[i]\\ &= -\sum_{j=1}^{d_{l+1}}\mathbf{W}_l[j,i]\frac{\partial\ell(\hat{\mathbf{y}},\mathbf{y})}{\partial\boldsymbol{h}_{l+1}[j]}\cdot g'(\mathbf{W}_l\boldsymbol{h}_l)[j]\cdot\boldsymbol{h}_l[i] && \text{(Using equation B.2)}\\ &= -\sum_{j=1}^{d_{l+1}}\mathbf{W}_l[j,i]\cdot\frac{\partial\ell(\hat{\mathbf{y}},\mathbf{y})}{\partial\mathbf{W}_l[j,i]}, && \text{(Using equation B.1)}\end{aligned}$$

which corresponds to the first-order utility in (4.5).