The effect of architecture during continual learning

Allyson Hahn *ahahn2813@gmail.com*
Mathematics and Computer Science
Argonne National Laboratory

Krishnan Raghavan* *kraghavan@anl.gov*
Mathematics and Computer Science
Argonne National Laboratory

Abstract

We investigate the role of neural network architecture in continual learning and identify that absolute continuity of the forgetting cost function is necessary to find a solution to the continual learning problem. To obtain the necessary condition, we introduce a mathematical framework that jointly models architecture and weights in a Sobolev function space. We then utilize this framework to characterize the coupling between model parameters and architectural choices and to analyze how changes in architecture affect the continuity of the forgetting loss. Using this insight, we demonstrate that training only the model weights is insufficient to mitigate catastrophic forgetting when data distributions evolve across tasks and architecture must be modified on the fly. To modify architecture while learning the weights, we formulate the continual learning as a bilevel optimization problem: the upper level selects an optimal architecture for a given task, while the lower level computes optimal weights through dynamic programming over all tasks.

To solve the upper-level problem, we introduce a derivative-free direct search algorithm that seeks to generate the best architecture by searching for the best architecture in the vicinity of the previous architecture. Once the architecture is found, we seek to transfer knowledge from the previous architecture to the new one. However, finding an optimal architecture on the fly would result in two parameter spaces with different and mismatched weight spaces. Therefore, we develop a low-rank transfer mechanism to map knowledge across architectures of mismatched dimensions. Empirical studies across regression and classification problems, including feedforward convolutional neural networks and graph neural networks, demonstrate that architectures optimized on the fly yield substantially improved performance (up to two orders of magnitude), reduced forgetting, and enhanced robustness in the presence of noise compared with static architecture approaches.

1 Introduction

With the advent of large-scale AI models, the computational expense of training such models has also increased drastically; training these models efficiently often requires large-scale high-performance computing infrastructure. Despite such drastic expenditure, these models quickly become stale. The reason is that the data distribution shifts or new data gets generated, requiring realignment. This necessity presents a quandary. Naive realignment leads to a phenomenon called catastrophic forgetting, where the model overwrites prior information. On the other hand, a complete retraining of the model would require significant resources, a solution that is extremely unattractive.

A more viable option is continual learning, where approaches seek to constrain the parameters of the model to reduce the amount of catastrophic forgetting the model experiences. Several works in the literature have attempted this direction, starting from early work on small multilayer perceptrons McCloskey & Cohen (1989) to recent works on large language models such as Luo et al. (2025); Lai et al. (2025); Biderman et al. (2024) and Lin et al. (2025). One key commonality across all these approaches is that they look to constrain the

weight/parameters of the AI model to induce minimal forgetting. On the other hand, and more appropriately, some approaches seek a balance between forgetting and the learning of new data (generalization); see, for example, Raghavan & Balaprakash (2021); Lu et al. (2025); Lin et al. (2023). Despite demonstrated success, this is not the full story. In fact, under mild assumptions, it has been proven that simply changing the weight of the AI model is not sufficient to capture the drift in the data distribution, and the capacity of the neural network (its ability to represent tasks) eventually diverges if the data distribution keeps changing Chakraborty & Raghavan (2025). These intractability results from Chakraborty & Raghavan (2025) highlight a conundrum: if the capacity of the network eventually diverges, then should we still aim to continually train AI models? In this paper we demonstrate that the answer to this question is indeed "yes"; in fact, *the solution is to reliably change the architecture of the AI model on the fly according to the needs of the data distribution.*

However, three key bottlenecks remain before attaining this goal. (i) Reliably changing the architecture requires a methodical understanding of "how the weights and the architecture of the model interact with each other," that is, the coupling between the weights and the architecture. (ii) Moreover, to decide when to change the architecture, one must understand how the forgetting-generalization trade-off is affected by the aforementioned coupling. Loosely speaking, the first two bottlenecks have been heuristically addressed in the literature. For example, some recent works such as CLEAS (Continual Learning with Efficient Architecture Search) Gao et al. (2022) and SEAL (Searching Expandable Architectures for Incremental Learning) Gambella et al. (2025) involve dynamically expanding the capacity of the AI model by adding layers and defining metrics that govern the necessity for a change in capacity. However, none of these approaches involves or enables a generic ability to change different aspects of the architecture, number of parameters, activation functions, layers, and so forth. The key reason is that if the architecture is generically changed, one must initialize the new network architecture at random. (iii) Then, the transfer of information about the weights from the previous architecture to the new architecture across parameter spaces of two different shapes is required. Doing so, however, is currently impossible. In the absence of such a transfer mechanism, current approaches such as CLEAS and SEAL can modify only those components that would not warrant information transfer between parameter space of two different sizes.

To address these bottlenecks, we present a comprehensive approach that includes a new formulation to understand the coupling between the architecture and the weights of the model, a methodical way of searching for a generic new architecture, and a novel method for transferring information between the old and the new architecture.

1.1 Contribution

The key reason the coupling between the architecture and the weights is difficult to model is that architecture dependencies are observed on a function space across tasks and the weight dependencies are visible across the Euclidean parameter space. Any modeling in the weight space such as presented in Chakraborty & Raghavan (2025); Raghavan & Balaprakash (2021) or Lu et al. (2025) cannot capture function space dependencies. To obviate this situation, we model the problem of continually training the AI model over a sequence of tasks in a Sobolev space. Using this theoretical framework, we describe how, in the Sobolev space of AI models parameterized by architecture choices and weights, weights alone cannot capture the change in the distribution: the architecture must be changed. We then employ a derivative-free architecture search to determine the new architecture by searching for an appropriate number of neurons. Although we focus on changing the number of parameters in the AI model, our work is general enough to extend to other architecture choices as well. Once the new architecture is chosen, we develop a new algorithm that can efficiently transfer information from the previous architecture to a new architecture across parameter spaces of different sizes with minimal loss of performance on the previous tasks.

We empirically demonstrate that changing the architecture indeed results in better training loss compared with when the architecture is not changed. We also show that our algorithm achieves substantial improvements in terms of training loss when training over large numbers of tasks; it is robust to noise and scales from feedforward neural networks to graph neural networks seamlessly across regression and classification problems. We envision that jointly training the architecture and the weights in the continual learning paradigm is the best path forward in the field of continual learning, and we develop the basic theoretical and empirical infrastructure to allow for such training.

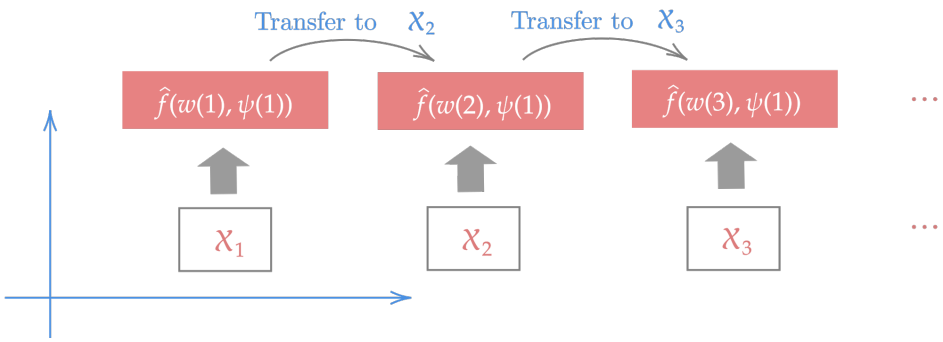


Figure 1: The basic problem of continual learning

1.2 Organization

The paper is organized as follows. We begin with a description of continual learning (CL) and motivate the necessity for function space modeling. Next, we describe a collection of neural networks as functions in a Sobolev space. We then derive the necessary condition for the existence of a CL solution and demonstrate that weights alone are not enough to train a network efficiently. Thus, we describe the need to change the architecture of the network along with the weights of the AI model. We present an algorithm to change the architecture on the fly and transfer information between the two architectures. We describe empirically the advantages of our approach and conclude with a brief summary.

1.3 Notation

The notation is adapted from Kolda & Bader (2009); please refer to the original paper for additional details. We use $\mathbb{N}$ to denote the set of natural numbers and $\mathbb{R}$ to denote the set of real numbers. An m^{th}-order tensor is viewed as a multidimensional array contained in $\mathbb{R}^{I_1 \times I_2 \times I_3 \times I_4 \times \cdots I_m}$, where the order can be thought of as the number of dimensions in the tensor. In this paper we will be concerned mostly with tensors of order $0, 1, 2$ and 3, which correspond to scalars, vectors, matrices, and a list of matrices. Therefore, we will write a tensor of order 0 as a scalar with lowercase letters, x; a tensor of order one is denoted by lowercase bold letters such as $\mathbf{x}$. A tensor of order 2 is denoted by uppercase bold letters $\mathbf{X}$, and any tensor of order greater than 2 is denoted by bolder Euler scripts letters such as $\boldsymbol{\mathcal{X}}$. We also use $\|.\|$ to denote the Euclidean norm for vectors, Frobenius norm for matrices, and an appropriate tensor norm for tensors. Further, we will use $\langle \cdot, \cdot \rangle$ to denote the dot product for vectors, matrices, or tensors. We will make one exception in our notation regarding the tensors that represent learnable/user-defined parameters (architecture, step-size/learning rate, etc.): we will denote these with Greek letters. The i^{th} element of a vector $\mathbf{x}$ is denoted by $\mathrm{x}_{[i]}$, while the $(i,j)^{th}$ element of a matrix $\mathbf{X}$ is denoted by $\mathrm{x}_{[ij]}$. Moreover, we denote the i^{th} matrix in a tensor of order 3 by $\mathbf{X}_i$. We will make the indices run from 1 to their capital letters such that $i = 1, \ldots, I$.

2 Continual Learning – Motivation

The problem of continual learning is that of an AI model learning a sequence of tasks. Here, we define a task as a set of data available to learn from. Without going into the specifics of the underlying space of an AI model, we will denote the AI model as $\hat{f}(\mathbf{w}, \psi)$, where f denotes the composition of the weights$-\mathbf{w}$ and the architecture$-\psi$. We will define the particulars of these quantities in a later section; however, the key objective of this model is to learn a series of tasks. In this context, each task is represented by a data set obtained at a task instance, that is, $t \in \boldsymbol{\mathcal{T}}, \boldsymbol{\mathcal{T}} = \{0, 1, \ldots, T\}$. We will assume that the dataset (a list of matrices, vectors, graphs) represented by $\boldsymbol{\mathcal{X}}(t)$ is provided at each task $t \in \boldsymbol{\mathcal{T}}$, where $\boldsymbol{\mathcal{X}}(t)$ is sampled according to the distribution $\mathbb{P}$ and $\boldsymbol{\mathcal{X}}(t) \subset \mathrm{D} \subset \mathbb{R}^n$ such that D is a measurable set with a non-empty interior. Moreover, $(\mathrm{D}, \mathcal{B}(\mathrm{D}), \mathbb{P})$ forms a probability triplet with $\mathcal{B}(\mathrm{D})$ being the Borel sigma algebra over the domain D. The problem of interest to us is to understand the behavior of the AI model when it is learning tasks $\boldsymbol{\mathcal{X}}(t)$ for every task instance t. Task instances can belong to $\mathbb{N}$ and $\mathbb{R}$ depending on the application at

hand, but the key goal is to understand how the tasks are assimilated by the AI model. The basic notion of continual learning is described in Figure 1, where there are three tasks at $t = 1, 2$, and 3. For each of these three tasks, an architecture $\psi(1)$ is chosen, and the three tasks are provided to the model in series. At $t = 1$, the model $\hat{f}$ learns from the first task. In particular, we solve the optimization problem

$$min_{\mathbf{w}(1)} \ell(\hat{f}(\mathbf{w}(1), \psi(1)), \boldsymbol{\mathcal{X}}(1)), \tag{1}$$

where ℓ is some loss function that summarizes the model's performance on the task at $t = 1$. Then, at $t = 2$, the model transfers to learn information from the second task such that the first task is not forgotten. Implicitly, the optimization problem is rewritten as

$$min_{\mathbf{w}(2)} \left[\ell(\hat{f}(\mathbf{w}(2), \psi(1)), \boldsymbol{\mathcal{X}}(1)) + \ell(\hat{f}(\mathbf{w}(2), \psi(1)), \boldsymbol{\mathcal{X}}(2)) \right]. \tag{2}$$

Similarly, at $t = 3$, the model transfers to learn information from the third task, but both tasks 1 and 2 must not be forgotten. This implies that

$$min_{\mathbf{w}(3)} \left[\ell(\hat{f}(\mathbf{w}(3), \psi(1)), \boldsymbol{\mathcal{X}}(1)) + \ell(\hat{f}(\mathbf{w}(3), \psi(1)), \boldsymbol{\mathcal{X}}(2)) + \ell(\hat{f}(\mathbf{w}(3), \psi(1)), \boldsymbol{\mathcal{X}}(3)) \right]. \tag{3}$$

As shown, the objective of the model is to remember all earlier tasks. This implies that the objective function of the continual learning problem is a sum that grows with every new task. Therefore, at any task t, the objective of the CL problem is

$$min_{\mathbf{w}(t)} \sum_{i=1}^{t} \left[\ell(\hat{f}(\mathbf{w}(i), \psi), \boldsymbol{\mathcal{X}}(i)) \right], \qquad \text{(Forgetting Loss)}$$

where we have removed the index from ψ to indicate that the architecture is fixed. In the CL literature, (Forgetting Loss) is typically optimized at the onset of every new task at t. The most rudimentary approach for this optimization involves an experience replay array that maintains a memory of all tasks in the past. To gain more insight into what happens at the onset of each new task, see Figure 2.

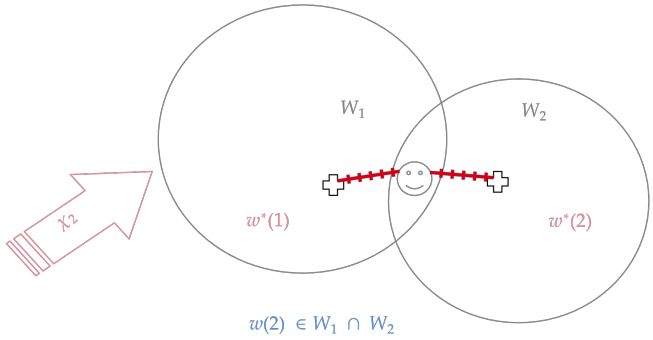


Figure 2: The basic problem of continual learning

For the sake of illustration, we assume that $\ell(\hat{f}(\mathbf{w}(i), \psi, \boldsymbol{\mathcal{X}}(i))$ is a strongly convex function described in Figure 2, the level set (the balls) in the parameter space corresponding to $\ell(\hat{f}(\mathbf{w}(i), \psi, \boldsymbol{\mathcal{X}}(i)) \leq \eta$ and the plus sign corresponding to the minima for $i = 1$ and $i = 2$. The threshold to identify acceptable loss values is η; it defines the boundary of the level set and, by extension, the radius of the ball in Figure 2. For instance, in the MNIST dataset, the two balls may represent two tasks representing digit 0 and digit 1. The boundary of the left ball represents all parameters that provide cross-entropy loss less than $\eta = 1$ for digit 0, with the plus sign representing zero cross-entropy loss. Similarly, the right ball represents all parameters that provide cross-entropy loss less than $\eta = 1$ for digit 1, and the plus sign corresponds to zero cross-entropy loss on digit 1. Within this example, consider an AI model (maybe a convolutional neural network) that trains on digit

0 and achieves zero loss on digit 0, that is, attains the plus sign in the parameter space. Then, the AI model starts training for the next task—digit 1. First, it starts from the plus sign achieved for digit 0; this implies that there is an inherent bias due to the local minima achieved for digit 0. Second, when the new task is observed, the model must converge to a solution that is common to both digit 0 and digit 1. This solution is trivial if digits 0 and 1 are identical or very close to each other. Since this is not the case, however, the solution must lie in the intersection between the two balls, that is, the region of the smiley face. Since the size of the intersection space depends on how different tasks 0 and 1 are, the following informally stated theorem is observed.

Theorem 1 *Fix the number of weight updates required to obtain the optimal value at each task. Assume that each subsequent task introduces a value of change into the model with the change described in some appropriate metric. Let ℓ be the Lipschitz continuous function, and choose a small learning rate. Then, capacity diverges as a function of the number of tasks.* ***Please see Chakraborty & Raghavan (2025) for a full statement and its proof.***

The theorem implies that the AI model's ability to successfully represent tasks will deteriorate as more tasks are introduced when the new tasks are different from the previous one. In such a case, continually training the AI model is bound to fail if we just update the weights of the network. In this paper we hypothesize that the size of the intersection space in Figure. 2 can be increased by modifying the architecture of the AI model, that is, by increasing the number of parameters in the model. Doing so, however, would require us to model the coupling between the weights, data, and the architecture, a task impossible to imagine with the intuition laid out until now since the notion of architecture cannot be modeled with this intuition. To this end, we will describe a neural network as a member of a class of functions in a function space that we choose to be a Sobolev space.

3 Neural Networks – Members of a Class of Sobolev Space Functions

The goal of this section is to formalize the mathematical modeling required to identify the dependency between the intersection space from Figure 2, the architecture and the AI model. We first define the AI model $\hat{f}(\mathbf{w}, \psi)$ to belong to a class of functions contained in a Sobolev space with k-bounded weak derivatives. The notion of k-bounded weak derivatives is the key reason that Sobolev space is an appropriate choice for this problem, which we will discuss once we outline the formal definitions of the Sobolev space aligning with definitions provided in Mahan et al.. **GAIL - I note that this reference in print does not have the date, as other citations in the text do.**

Definition 2 *Let $k \in \mathbb{N}$, the domain-$\mathrm{D} \subset \mathbb{R}^n$ a measurable set with non-empty interior, and $1 < p < \infty$. Then the Sobolev space $W^{k,p}(\mathrm{D})$ consists of all functions f on D such that for all multi-indices α with $|\alpha| \leq k$, the mixed partial derivative $\partial^{(\alpha)} f$ exists in the weak sense and belongs to $L^p(\mathrm{D})$. That is,*

$$W^{k,p}(\mathrm{D}) = \{f \in L^p(\mathrm{D}) : \partial^{|\alpha|} h \in L^p(\mathrm{D}) \forall |\alpha| \leq k\}.$$

The number k is the order of the Sobolev space, and the Sobolev space norm is defined as

$$\|f\|_{W^{k,p}(\mathrm{D})} := \sum_{|\alpha| \leq k} \|\partial^{|\alpha|} f\|_{L^p(\mathrm{D})}.$$

In Definition 2, two key notions must be highlighted. First, the D is by definition a subset of $\mathbb{R}^n$, which by virtue of the Heine–Borel theorem of Rudin (1976) is compact and closed. The implications of this assumption are important. D, in this context, is the data sample space, and therefore the assumption of compact and closedness applies to the data space. For any standard AI application, this assumption implies that the data space is finite and the numerical values are bounded. Both are notable practical considerations that must be followed in any AI model training as it is well known that without the normalization, the AI model training fails to converge Huang et al. (2023). GAIL - Here and throughout, this way of referencing seems a bit odd – having the author name followed by parens and the date but not separation from the text

Name	$\rho(\mathbf{x})$	Smoothness /Boundedness	Corresponding $W^{k,p}$
Rectified Linear Unit (ReLU)	$sup\{0, \mathbf{x}\}$	absolutely continuous, $\rho' \in L^p(\mathrm{D})$	$W^{0,p}(\mathrm{D})$
Exponential Linear Unit (eLU)	$\mathbf{x}.\chi_{\mathbf{x}\geq 0}$ $+(e^{\mathbf{x}}-1)\chi_{\mathbf{x}<0}$	$C^1(\mathbb{R})$, absolutely continuous $\rho'' \in L^p(\mathrm{D})$	$W^{1,p}(\mathrm{D})$
Softsign	$\frac{\mathbf{x}}{1+\lvert\mathbf{x}\rvert}$	$C^1(\mathbb{R})$, ρ' absolutely continuous, $\rho'' \in L^p(\mathrm{D})$	$W^{2,p}$
Inverse Square Root Linear Unit	$\mathbf{x}\chi_{\mathbf{x}\geq 0}$ $+\frac{\mathbf{x}}{\sqrt{1+a\mathbf{x}^2}}\chi_{\mathbf{x}<0}$	$C^2(\mathbb{R})$, ρ'' absolutely continuous, $\rho''' \in L^p(\mathrm{D})$	$W^{3,p}$
Inverse Square Root Unit	$\frac{\mathbf{x}}{\sqrt{1+a\mathbf{x}^2}}$	real analytic, real, all derivatives bounded	$W^{k,p}\forall k$
Sigmoid	$\frac{1}{1+e^{\mathbf{x}}}$	real analytic, real, all derivatives bounded	$W^{k,p}\forall k$
tanh	$\frac{e^{\mathbf{x}}-e^{-\mathbf{x}}}{e^{\mathbf{x}}+e^{-\mathbf{x}}}$	real, analytic, real, all derivatives bounded	$W^{k,p}\forall k$

Table 1: Different activation functions and the corresponding Sobolev spaces. Many of these results are described in Petersen et al. and Adegoke & Layeni . χ is an indicator function.

– usually one has (Author year) or "as discussed by author (year), not just "fails to converge Huang et al. (2023)" – as if Huang et al. wee failing to converge!

Therefore, this assumption is not only practical but necessary.

The next assumption is the presence of weak derivatives. In a standard neural network, derivatives are assumed to exist since twice differentiability is necessary for training and typical training involves gradient-based methods Kingma & Ba. In fact, the existence of weak derivatives can be summarized for different activation functions as in Table 1. As noted in the table, all standard activation functions can be recast in the purview of Definition 2, and a Sobolev space can be constructed that represents the class of functions that can be approximated by neural networks. The appropriateness of approximation and the conditions over which such approximations are possible are discussed in Petersen et al. and Adegoke & Layeni.

To represent the class of functions described in Definition 2, we define a neural network as a function $\hat{f}(\mathbf{w}(t), \psi(t))$ with $d \in \mathbb{N}$ layers. In this notation, $\mathbf{w}(t)$ is a long $\mathbb{R}^m$ vector that concatenates the weight parameters from the d layers, and $\psi(t)$ comprises the architecture and other user-defined parameters in the neural network.

Definition 3 *A d-layered neural network is given by an operator (essentially a function)* $\hat{f}(\mathbf{w}(t), \psi(t)) \in W^{k,p}(\mathrm{D})$ *with* $W^{k,p}(\mathrm{D})$ *being a Sobolev space. Furthermore,*

$$\hat{f}(\mathbf{w}(t), \psi(t))\big(.\big) = \hat{f}_d(\mathrm{w}_d(t), \psi_d(t)) \circ \hat{f}_{d-1}(\mathrm{w}_{d-1}(t), \psi_{d-1}(t)) \circ \cdots \circ \hat{f}_2(\mathrm{w}_2(t), \psi_2(t)) \circ \hat{f}_1(\mathrm{w}_1(t), \psi_1(t))\big(.\big) \quad (4)$$

describes the layer-wise compositions, and $\big(.\big)$ *represents the input tensor to which the operator is applied.*

Remark 1 *We indicate approximate quantities with* $\hat{.}$ *and true (target) quantities without the hat. For instance, the target function* f *in Definition 2 is indicated without the hat, and the approximate function* $\hat{f}$ *in definition 3 is indicated with a hat.*

Remark 2 *In Definition 3 we describe the operation across different layers using the composition; in terms of notation, we hide this complexity through the definition of the operator. Therefore, Definition 3 is generic and can be used to define feedforward, recurrent, convolutional, and graph neural networks and even a large language model. Therefore, any analysis from this point is applicable to all types of neural networks.*

Typically, the user-defined parameters are a combination of integer, categorical, and floating-point values; and we assume that the architecture can be varied with task instances. Therefore, the parameters corresponding to the architecture ψ are assumed to be a function of the task instance t. Since we cannot determine derivatives with respect to discrete parameters in the classical sense (i.e., derivative of $\psi(t)$), we need to define weak derivative with respect to these quantities such that the derivatives (with respect to or of ψ) can be approximated. To describe the learning problem, we define a notion of loss function on the space of functions defined in Definition 3 and 2.

Definition 4 *Let $f(t) \in W^{k,p}(\mathrm{D})$ for all $t \in \{0, \dots, T\}$ denote the target function that is to be approximated, and let $\hat{f}(\mathbf{w}(t), \psi(t))$ denote a neural network determined to approximate $f(t)$ at task t. Then the loss function (or error of approximation) for $\mathbf{x} \in \boldsymbol{\mathcal{X}}(t) \subseteq \mathrm{D}$ is given by*

$$\ell(\hat{f}(\mathbf{w}(t), \psi(t)), \mathbf{x}) = \|\hat{f}(\mathbf{w}(t), \psi(t))(\mathbf{x}) - f(t)(\mathbf{x})\|_{W^{k,p}(\mathrm{D})}.$$

The function f in the context of machine learning is an ideal forecasting model that can forecast temperature perfectly or an ideal classification model that performs some classification problem.

Remark 3 *In applications, we assume* D *is compact to guarantee that the target function $f(t)$ exists in a Sobolev space $W^{k,p}(\mathrm{D})$, by Corollary 3.5 in Mahan et al.. If, however, compactness cannot be assumed, we find the minimum norm solution to the problem in the Sobolev space.*

The loss function in Definition 4 is defined over $W^{k,p}(\mathrm{D})$ using the Sobolev space norm. In practice, however, we can directly utilize samples corresponding to f or $\hat{f}$ to construct the loss function. For instance, this has been done in Son et al. (2021) in the context of physics-informed neural networks—a class of neural networks modeled by Definition 3. In particular, the Sobolev space norm is defined as in Definition 2, where the loss function is a sum over L^p norm over all k weak derivatives. As a trivial case, choosing $k = 0$ and $p = 2$ gives the standard mean squared error loss function that, according to Table 1 and practical consideration, is applicable to various activation functions used in practical AI.

Continual Learning in $W^{k,p}(\mathrm{D})$

We extend Definition 4 to the case of CL with the goal of learning a series of tasks. In particular, for each task instance t we must find a $\mathbf{w}^*$ with respect to ℓ integrated over the interval $[0, t]$. This objective function denoted J is known as the forgetting loss and was informally defined as

$$J(\mathbf{w}(t), \psi(t), \boldsymbol{\mathcal{X}}(t)) = \int_0^t \left(\int_{\mathbf{x} \in \boldsymbol{\mathcal{X}}(\tau)} \ell(\hat{f}(\mathbf{w}(t), \psi(t))(\mathbf{x})) \right) \, d\, \tau.$$

A traditional learning structure as in Pasunuru & Bansal (2019) seeks to minimize the cost $J(t)$ through an optimizer (one such as Adam Kingma & Ba) to drive $J(t) \to 0$ when $t \to \infty$, a structure mathematically guaranteed to converge, as shown in Kingma & Ba. In practice, however, when consecutive tasks are very different from each other, we encounter the issue described in Figure 2. In order to resolve this quandary, for each new task, the underlying optimization problem must be repeatedly resolved, leading to a series of optimization problems, altogether forming a dynamic program Bertsekas (2012). A standard practice in the optimal control literature is to solve such dynamic programs Bertsekas (2012) through a cumulative optimization problem over the interval $[0, T]$ as in

$$V^*(t, \mathbf{w}(t)) = \min_{\mathbf{w} \in \mathcal{W}(\psi^*(t))} V(t, \mathbf{w}(t)),$$

where $\mathcal{W}(\psi^*(t))$ is the weights search space and $V(t, \mathbf{w}(t))$ is

$$V(t, \mathbf{w}(t)) = \int_t^T J(\mathbf{w}(\tau), \psi^*(t), \boldsymbol{\mathcal{X}}(\tau)).$$

An alternative learning process that solves this dynamic program was introduced in Raghavan & Balaprakash (2021); Chakraborty & Raghavan (2025) that uses a fixed architecture $\psi^*(t)$. However, as argued in Section 2, fixed architecture may not be enough in the CL paradigm. Therefore, we are interested in a scenario where the architecture and the model parameters are both learned. To achieve this, we must complete a bi-level optimization with the bottom level being a dynamic program. The upper problem is outlined as follows:

$$\psi^*(t) = \arg\min_{\psi \in \Psi} J(\mathbf{w}(t), \psi, \boldsymbol{\mathcal{X}}(t)), \qquad \text{(Architecture search)}$$

where Ψ is our architecture search space. Once the search is completed, we move to the lower problem to find the weights as follows:

$$V^*(t, \mathbf{w}(t)) = \min_{\mathbf{w} \in \mathcal{W}(\psi^*(t))} V(t, \mathbf{w}(t)). \qquad \text{(Cumulative CL)}$$

As discussed earlier, naively solving this problem will involve repeatedly retraining the model from scratch for each new task and each new architecture. This is both computationally and mathematically unattractive.

4 Understanding the Existence of the CL Solution

In this paper we will devise an alternative approach to solving the bi-level problem. First, however, we must understand the dependence between the architecture, weights, and the Forgetting Loss. To understand this behavior, let us reconsider the example from Section 2 and redraw Figure 3.

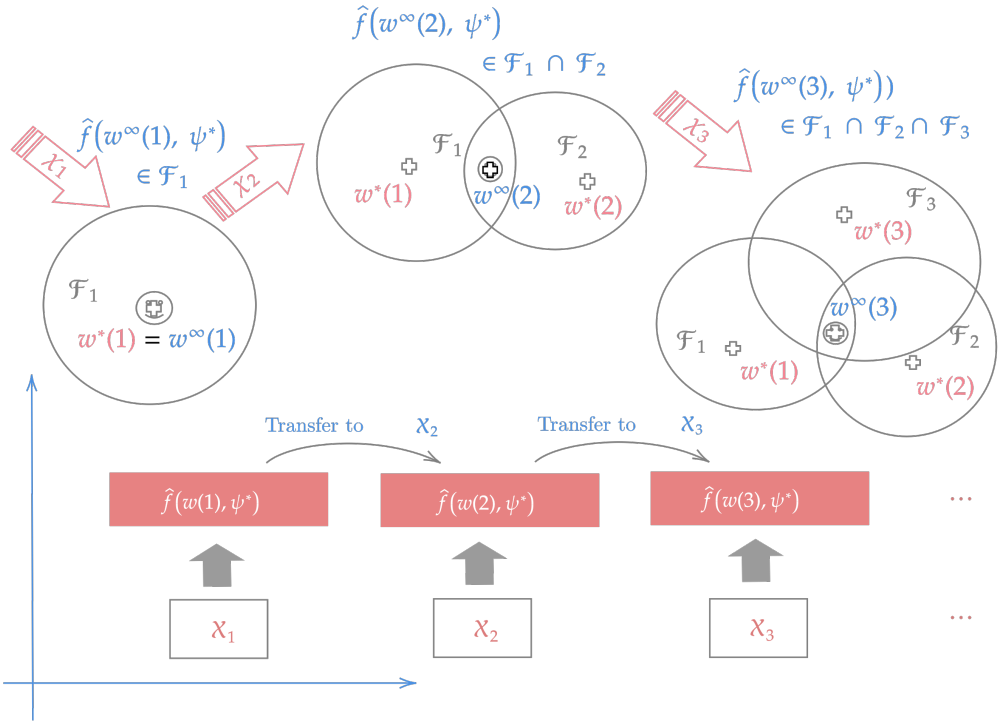


Figure 3: The actual problem

From Figure 3, let $\mathcal{F}_t = \{\hat{f}(\mathbf{w}, \psi) \in W^{k,p}(\mathrm{D})\}$ represent the search space for the set of all possible neural network solutions for learning task t (refer to 2 for a similar construction). Since $\mathcal{F}_t$ is a subset of $W^{k,p}(\mathrm{D})$,

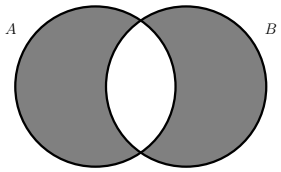


Figure 4: Set symmetric difference $A \triangle B$ represented by the gray-shaded region

the balls in Figure 3 can be thought of as a level set such that the boundary of the ball represents the set of all solutions that provide the loss value less than a threshold (a threshold set by the user). At $t = 1$ there is one ball, at task $t = 2$ there are two balls, and at task $t = 3$ there are three balls. The number of balls here corresponds to the number of tasks involved in CL.

In each of the balls, ✚ represents the neural network solution for just task t, and the ☺ face represents the solution that performs well among tasks between $0 \rightarrow t$. Notably, ☺ must lie in the intersection of all the balls from 0 to t. Note that unless the tasks are equivalent, ☺ and the ✚ will not coincide because no solution that is perfect for one task can be perfect for another. Furthermore, the more different the subsequent tasks are, the more likely the intersection space is to be empty, and the CL problem has no solution across all tasks from 0 to t. From a different perspective, the non-emptiness of the intersection space implies that the union of all the balls is a connected set from a topological perspective. A continuous function such as J, defined on this connected set, will have a gradient that is defined completely over the union of these balls. Therefore, if the intersection space is empty, the connectivity of the union of these balls is violated, and the gradient of J is undefined, which will lead to a solution that cannot be found. The best one can do in this circumstance is a minimum norm solution as is done in the current literature.

Mathematically, this intuition is described by the notion of continuity of J in the closed interval $[t, t+1]$ (the union of any two balls) and the absolute continuity of J over the interval $[0, T]$ (the union of all balls in a complete interval) since the notion of these balls is defined dependent on the tasks. We will define measure μ over the task probability space $\mathbb{P}$ and describe an ε, δ definition of continuity of a function w.r.t this measure defined over the set symmetric difference $\mu(A \triangle B)$.

Definition 5 *Let $\overline{\boldsymbol{\mathcal{X}}} = \bigcup_{t=0}^{\infty} \boldsymbol{\mathcal{X}}(t)$ with the power set $\mathcal{P}(\overline{\boldsymbol{\mathcal{X}}})$ as its topology. Set $\mathcal{B}(\overline{\boldsymbol{\mathcal{X}}})$ to be the Borel sigma algebra on $\overline{\boldsymbol{\mathcal{X}}}$ equipped with a probability measure denoted μ. Then $(\overline{\boldsymbol{\mathcal{X}}}, \mathcal{B}(\overline{\boldsymbol{\mathcal{X}}}), \mu)$ forms a probability measure space. Let $\hat{f} : \overline{\boldsymbol{\mathcal{X}}} \rightarrow \mathbb{R}$ be a measurable function, and set $F : \mathcal{B}(\overline{\boldsymbol{\mathcal{X}}}) \rightarrow \mathbb{R}$ to be the function*

$$F(A) = \int_{\mathbf{x} \in A} \hat{f}(\mathbf{x}) d\mu,$$

where $\hat{f}(\mathbf{w}(t), \psi(t))(\mathbf{x}) = \hat{f}(\mathbf{x})$. Then we say that F is continuous with respect to the measure if for every $\varepsilon > 0$ there exists $\delta > 0$ such that $|F(A) - F(B)| < \varepsilon$ whenever $A, B \in \mathcal{B}(\overline{\boldsymbol{\mathcal{X}}})$ such that $\mu(A \triangle B) < \delta$.

Definition 5 can be deduced from basic measure theory results found in Weaver (2013) implying that sets that are similar (with respect to the probability measure μ) result in close values of $F(A)$, integrated over the respective sets. In particular, $\mu(A \triangle B)$ is the Lebesgue measure μ applied over $A \triangle B$ representing a set symmetric difference, which describes the elements two sets do not share (see Figure 4 for an illustration). Therefore, as the intersection space (i.e., $\mu(A \cap B)$) shrinks, $\mu(A \triangle B)$ gets larger, and $|F(A) - F(B)|$ gets larger proportionally, thus providing a measure of continuity that is proportional to the number of common elements between the two sets. More common elements imply a smaller value of the measure, and less common elements imply a large value of the measure. It is straightforward to see that $\mu(A \triangle B)$ is a Radon–Nikodym measure derived from the probability measure μ.

4.1 Existence

To begin, we prove that the expected value function, that is, $\int_{\mathbf{x} \in A} \ell(\hat{f}(\mathbf{w}, \psi)(\mathbf{x})) d\mu$, is continuous with respect to measure μ when $\mu(A \triangle B) < \delta, \forall A, B \in \overline{\boldsymbol{\mathcal{X}}}$.

Lemma 6 *Let $(\overline{\boldsymbol{\mathcal{X}}}, \mathcal{B}(\overline{\boldsymbol{\mathcal{X}}}), \mu)$ be the measure space as defined in Definition 5. Assume that the loss function ℓ is continuous and bounded across all tasks, that is, $\forall t \in [0, T]$. Then the expected value function*

$$E(A) = \int_{\mathbf{x} \in A} \ell(\hat{f}(\mathbf{w}, \psi)(\mathbf{x}))d\mu. \tag{5}$$

$$\tag{6}$$

is continuous with respect to measure μ.

Proof 1 *Suppose that the constant $M_0 > 0$ is the value that bounds ℓ on every task. Let $\varepsilon > 0$, and set $\delta = \varepsilon / M_0$. Further, let $A, B \in \mathcal{B}(\boldsymbol{\mathcal{X}})$ such that $\mu(A \triangle B) < \delta$. By disjoint additivity of μ and the triangle inequality, we have*

$$\begin{aligned}
|E(A) - E(B)| &= |\int_{\mathbf{x} \in A} \ell(\hat{f}(\mathbf{w}, \psi)(\mathbf{x}))d\mu - \int_{\mathbf{x} \in B} \ell(\hat{f}(\mathbf{w}, \psi)(\mathbf{x}))d\mu| \\
&= |\int_{\mathbf{x} \in A \setminus B} \ell(\hat{f}(\mathbf{w}, \psi)(\mathbf{x}))d\mu + \int_{\mathbf{x} \in A \cap B} \ell(\hat{f}(\mathbf{w}, \psi)(\mathbf{x}))d\mu \\
&- \int_{\mathbf{x} \in B \setminus A} \ell(\hat{f}(\mathbf{w}, \psi)(\mathbf{x}))d\mu - \int_{\mathbf{x} \in A \cap B} \ell(\hat{f}(\mathbf{w}, \psi)(\mathbf{x}))d\mu| \\
&= |\int_{\mathbf{x} \in A \setminus B} \ell(\hat{f}(\mathbf{w}, \psi)(\mathbf{x}))d\mu - \int_{\mathbf{x} \in B \setminus A} \ell(\hat{f}(\mathbf{w}, \psi)(\mathbf{x}))d\mu| \\
&\leq \int_{\mathbf{x} \in A \setminus B} |\ell(\hat{f}(\mathbf{w}, \psi)(\mathbf{x}))|d\mu + \int_{\mathbf{x} \in B \setminus A} |\ell(\hat{f}(\mathbf{w}, \psi)(\mathbf{x}))|d\mu.
\end{aligned}$$

Then, by the boundedness of ℓ, we have

$$\int_{\mathbf{x} \in A \setminus B} |\ell(\hat{f}(\mathbf{w}, \psi)(\mathbf{x}))|d\mu + \int_{\mathbf{x} \in B \setminus A} |\ell(\hat{f}(\mathbf{w}, \psi)(\mathbf{x}))|d\mu \leq M_0 \mu(A \setminus B) + M_0 \mu(B \setminus A).$$

Hence,

$$\begin{aligned}
M_0 \mu(A \setminus B) + M_0 \mu(B \setminus A) &= M_0 \mu(A \triangle B) \\
&< M_0 \delta \\
&= M_0 \frac{\varepsilon}{M_0} \\
&= \varepsilon,
\end{aligned}$$

as desired.

Now we show that two tasks $\boldsymbol{\mathcal{X}}(t), \boldsymbol{\mathcal{X}}(t+1)$ when similar (connected sets) such that $\mu(\boldsymbol{\mathcal{X}}(t) \triangle \boldsymbol{\mathcal{X}}(t+1)) < \delta$ imply that $|J(\mathbf{w}(t), \psi(t), \boldsymbol{\mathcal{X}}(t)) - J(\mathbf{w}(t+1), \psi(t+1), \boldsymbol{\mathcal{X}}(t+1))| < \varepsilon$ for a small $\varepsilon > 0$ with a $\delta > 0$. We prove this result in the following lemma.

Lemma 7 *Let $(\overline{\boldsymbol{\mathcal{X}}}, \mathcal{B}(\overline{\boldsymbol{\mathcal{X}}}), \mu)$ be the measure space as defined in Definition 5. Assume that the loss function ℓ is continuous and bounded across all tasks, that is, $\forall t \in [0, T]$. Define J over a task $\boldsymbol{\mathcal{X}}(t)$ as*

$$J(\mathbf{w}(t), \psi(t), \boldsymbol{\mathcal{X}}(t)) = \int_0^t \left(\int_{\mathbf{x} \in \boldsymbol{\mathcal{X}}(\tau)} \ell(\hat{f}(\mathbf{w}(t), \psi(t))(\mathbf{x})) \right) d\,\tau = \int_{\mathbf{x} \in \bigcup_{\tau=0}^{t} \boldsymbol{\mathcal{X}}(\tau)} \ell(\hat{f}(\mathbf{w}(\tau), \psi(\tau))(\mathbf{x}))d\tau. \tag{7}$$

Let $\varepsilon > 0$ and $\delta(\varepsilon) > 0$ chosen such that $\left(\int_{\mathbf{x} \in \boldsymbol{\mathcal{X}}(\tau)} \ell(\hat{f}(\mathbf{w}(t), \psi(t))(\mathbf{x})) \right)$ is continuous w.r.t. measure μ. Then, if $\mu(\bigcup_{\tau=0}^{t+\Delta t} \boldsymbol{\mathcal{X}}(\tau) \triangle \bigcup_{\tau=0}^{t} \boldsymbol{\mathcal{X}}(\tau)) < \delta$ implies $|J(\mathbf{w}(t+1), \psi(t+1), \boldsymbol{\mathcal{X}}(t+1)) - J(\mathbf{w}(t), \psi(t), \boldsymbol{\mathcal{X}}(t))| \leq \varepsilon$ with $\Delta t = 1$. GAIL - I don't follow the preceding: If XXX less than delta implies YYY – where is the If... then part?

Proof 2 *Suppose that the constant $M_0 > 0$ bounds ℓ on every task. Let $\varepsilon > 0$, and set $\delta = \varepsilon / M_0$. By disjoint additivity of μ and triangle inequality, we have*

$$\begin{aligned}
&\left| \int_0^{t+\Delta t} \left(\int_{\mathbf{x} \in \boldsymbol{\mathcal{X}}(\tau)} \ell(\hat{f}(\mathbf{w}(t), \psi(t))(\mathbf{x})) \right) d\mu \; d\; \tau - \int_0^{t} \left(\int_{\mathbf{x} \in \boldsymbol{\mathcal{X}}(\tau)} \ell(\hat{f}(\mathbf{w}(t), \psi(t))(\mathbf{x})) \right) d\mu \; d\; \tau \right| \\
&= \left| \int_{\bigcup_{\tau=0}^{t+\Delta t} \boldsymbol{\mathcal{X}}(\tau)} \ell(\hat{f}(\mathbf{w}, \psi)) d\mu - \int_{\bigcup_{\tau=0}^{t} \boldsymbol{\mathcal{X}}(\tau)} \ell(\hat{f}(\mathbf{w}, \psi)) d\mu \right| \\
&= \Big| \int_{\bigcup_{\tau=0}^{t+\Delta t} \boldsymbol{\mathcal{X}}(\tau) \setminus \bigcup_{\tau=0}^{t} \boldsymbol{\mathcal{X}}(\tau)} \ell(\hat{f}(\mathbf{w}, \psi)) d\mu + \int_{\bigcup_{\tau=0}^{t} \boldsymbol{\mathcal{X}}(\tau) \cap \bigcup_{\tau=0}^{t+\Delta t} \boldsymbol{\mathcal{X}}(\tau)} \ell(\hat{f}(\mathbf{w}, \psi)) d\mu \\
&- \int_{\bigcup_{\tau=0}^{t+\Delta t} \boldsymbol{\mathcal{X}}(\tau)} \ell(\hat{f}(\mathbf{w}, \psi)) d\mu - \int_{\bigcup_{\tau=0}^{t} \boldsymbol{\mathcal{X}}(\tau) \cap \bigcup_{\tau=0}^{t+\Delta t} \boldsymbol{\mathcal{X}}(\tau)} \ell(\hat{f}(\mathbf{w}, \psi)) d\mu \Big| \\
&= \Big| \int_{\bigcup_{\tau=0}^{t+\Delta t} \boldsymbol{\mathcal{X}}(\tau) \setminus \bigcup_{\tau=0}^{t} \boldsymbol{\mathcal{X}}(\tau)} \ell(\hat{f}(\mathbf{w}, \psi)) d\mu - \int_{\bigcup_{\tau=0}^{t+\Delta t} \boldsymbol{\mathcal{X}}(\tau)} \ell(\hat{f}(\mathbf{w}, \psi)) d\mu \Big| \\
&\leq \int_{\bigcup_{\tau=0}^{t+\Delta t} \boldsymbol{\mathcal{X}}(\tau) \setminus \bigcup_{\tau=0}^{t} \boldsymbol{\mathcal{X}}(\tau)} |\ell(\hat{f}(\mathbf{w}, \psi))| d\mu - \int_{\bigcup_{\tau=0}^{t+\Delta t} \boldsymbol{\mathcal{X}}(\tau)} |\ell(\hat{f}(\mathbf{w}, \psi))| d\mu
\end{aligned}$$

.

Then, by the boundedness of ℓ,

$$\begin{aligned}
&\int_{\bigcup_{\tau=0}^{t+\Delta t} \boldsymbol{\mathcal{X}}(\tau) \setminus \bigcup_{\tau=0}^{t} \boldsymbol{\mathcal{X}}(\tau)} |\ell(\hat{f}(\mathbf{w}, \psi))| d\mu - \int_{\bigcup_{\tau=0}^{t+\Delta t} \boldsymbol{\mathcal{X}}(\tau)} |\ell(\hat{f}(\mathbf{w}, \psi))| d\mu \\
&\leq M_0 \mu(\bigcup_{\tau=0}^{t+\Delta t} \boldsymbol{\mathcal{X}}(\tau) \setminus \bigcup_{\tau=0}^{t} \boldsymbol{\mathcal{X}}(\tau)) + M_0 \mu(\bigcup_{\tau=0}^{t} \boldsymbol{\mathcal{X}}(\tau) \setminus \bigcup_{\tau=0}^{t+1} \boldsymbol{\mathcal{X}}(\tau)).
\end{aligned}$$

Hence,

$$\begin{aligned}
M_0 \mu(\bigcup_{\tau=0}^{t+\Delta t} \boldsymbol{\mathcal{X}}(\tau) \setminus \bigcup_{\tau=0}^{t} \boldsymbol{\mathcal{X}}(\tau)) + M_0 \mu(\bigcup_{\tau=0}^{t} \boldsymbol{\mathcal{X}}(\tau) \setminus \bigcup_{\tau=0}^{t+1} \boldsymbol{\mathcal{X}}(\tau)) &= M_0 \mu(\bigcup_{\tau=0}^{t+\Delta t} \boldsymbol{\mathcal{X}}(\tau) \triangle \bigcup_{\tau=0}^{t} \boldsymbol{\mathcal{X}}(\tau)) \\
&< M_0 \delta \\
&= M_0 \frac{\varepsilon}{M_0} \\
&= \varepsilon,
\end{aligned}$$

as desired with $\Delta t = 1$.

Remark 4 *It is reasonable to assume that there is some constant $M_0 < \infty$ for which $M_i \leq M_0$ for all $i \in \mathbb{N}$ because if one did not exist, then the problem would be unsolvable since an unbounded loss function would imply that the neural network is not learning anything useful from the data.*

Remark 5 *The equality $\int_0^t \left(\int_{\mathbf{x} \in \boldsymbol{\mathcal{X}}(\tau)} \ell(\hat{f}(\mathbf{w}(t), \psi(t))(\mathbf{x})) \right) d\; \tau = \int_{\mathbf{x} \in \bigcup_{\tau=0}^{t} \boldsymbol{\mathcal{X}}(\tau)} \ell(\hat{f}(\mathbf{w}(\tau), \psi(\tau))(\mathbf{x})) d\tau$ is not always guaranteed because the duplicates among all $\boldsymbol{\mathcal{X}}(\tau), \forall \tau$ are counted only once. However, we assume that within the construction of ℓ, there is an average between all duplications, and only one instance of these duplicates survives. This is reasonable because, in practice, a batch of data is uniformly sampled and averaged over.*

Lemma 6 solidifies the notion that similar tasks produce similar loss values and that $J(\mathbf{w}(t), \psi(t), \boldsymbol{\mathcal{X}}(t))$ is continuous with respect to the measure μ for any $\boldsymbol{\mathcal{X}}(t), \boldsymbol{\mathcal{X}}(t+1) \in \mathcal{B}(\overline{\boldsymbol{\mathcal{X}}})$. The lemma provides the conditions for when a solution between two consecutive tasks exists. If Lemma 6 is upheld for every $t \in [0, T]$,, then the CL problem will have a solution for the whole interval $[0, T]$. To provide a rigorous notion of this, we have the following definition.

Definition 8 *Consider the interval* $[0,T]$ *and* $\overline{\boldsymbol{\mathcal{X}}} = \bigcup_{t=0}^{T} \boldsymbol{\mathcal{X}}(t)$ *with the power set* $\mathcal{P}(\overline{\boldsymbol{\mathcal{X}}})$ *as its topology. Set* $\mathcal{B}(\overline{\boldsymbol{\mathcal{X}}})$ *to be the Borel sigma algebra on* $\overline{\boldsymbol{\mathcal{X}}}$ *equipped with a probability measure* μ*. Then,* $(\overline{\boldsymbol{\mathcal{X}}}, \mathcal{B}(\overline{\boldsymbol{\mathcal{X}}}), \mu)$ *forms a probability measure space. Define the set* $[0,T] = [0,1] \cup (1,2], \cdots (t, t+1], \cdots, \cup, \cdots, \cup, (T-1, T]$*. Let* $\hat{f} : \overline{\boldsymbol{\mathcal{X}}} \to \mathbb{R}$ *be a measurable function. For any* $t \in [0,T]$*, define* $F(t) : \mathcal{B}(\overline{\boldsymbol{\mathcal{X}}}) \to \mathbb{R}$ *such that* $F(t) = \int_0^t \left(\int_{\mathbf{x} \in \boldsymbol{\mathcal{X}}(\tau)} \ell(\hat{f}(\mathbf{w}(t), \psi(t))(\mathbf{x})) \right) \, d\, \tau$*. If* $\sum_t^T \mu((\boldsymbol{\mathcal{X}}(\tau+1) \cup \boldsymbol{\mathcal{X}}(\tau)) \triangle \boldsymbol{\mathcal{X}}(\tau+1)) < \delta$ *implies* $\sum_t^T |F(t) - F(t+1)| \leq \varepsilon$*, then* F *is absolutely continuous over the interval* $[t,T]$*.*

With this definition and the notation that $E(\boldsymbol{\mathcal{X}}(\tau)) = \left(\int_{\mathbf{x} \in \boldsymbol{\mathcal{X}}(\tau)} \ell(\hat{f}(\mathbf{w}(t), \psi(t))(\mathbf{x})) \right)$ we have the following theorem.

Theorem 9 *Let* $(\overline{\boldsymbol{\mathcal{X}}}, \mathcal{B}(\overline{\boldsymbol{\mathcal{X}}}), \mu)$ *be the measure space as defined in Definition 5 and requirements for absolute continuity defined as in Definition 8. Assume that the loss function* ℓ *is continuous and bounded for all tasks, that is,* $\forall t \in [0,T]$*. Define* J *over a task* $\boldsymbol{\mathcal{X}}(\tau)$ *as*

$$J(\mathbf{w}(t), \psi(t), \boldsymbol{\mathcal{X}}(t)) = \int_0^t E(\boldsymbol{\mathcal{X}}(\tau)) \, d\, \tau. \tag{8}$$

Then $J(\mathbf{w}(t), \psi(t), \boldsymbol{\mathcal{X}}(t))$ *is absolutely continuous with respect to the measure for all* $\{\boldsymbol{\mathcal{X}}(\tau) \in \mathcal{B}(\overline{\boldsymbol{\mathcal{X}}}), \tau = 0, 1, \cdots, t\}$*.*

Proof 3 *Suppose that the constant* $M_0 > 0$ *bounds* ℓ *on every task. Let* $\varepsilon > 0$*, and set* $\delta = \varepsilon / T M_0$*. Furthermore, let* $\boldsymbol{\mathcal{X}}(t+1), \boldsymbol{\mathcal{X}}(t) \in \mathcal{B}(\overline{\boldsymbol{\mathcal{X}}})$ *such that* $\mu(\bigcup_{\tau=0}^{t+\Delta t} \boldsymbol{\mathcal{X}}(\tau) \triangle \bigcup_{\tau=0}^{t} \boldsymbol{\mathcal{X}}(\tau)) < \delta$*. Then, by the boundedness of* ℓ *and from Lemma 6 and Theorem 9 with the choice of* $\delta = \frac{\varepsilon}{2(T-t)M_0}$*, we have*

$$\begin{aligned}
&\sum_t^T [|J(\mathbf{w}(\tau + \Delta\tau), \psi(\tau + \Delta\tau), \boldsymbol{\mathcal{X}}(\tau + \Delta\tau)) - J(\mathbf{w}(t), \psi(t), \boldsymbol{\mathcal{X}}(t))|] \\
&\leq \sum_t^T \left| M_0 \mu(\bigcup_{\tau=0}^{t+\Delta t} \boldsymbol{\mathcal{X}}(\tau) \triangle \bigcup_{\tau=0}^{t} \boldsymbol{\mathcal{X}}(\tau)) \right| + \left| M_0 \mu \left(\bigcup_{\tau=0}^{t+\Delta t} \boldsymbol{\mathcal{X}}(\tau) \triangle \bigcup_{\tau=0}^{t} \boldsymbol{\mathcal{X}}(\tau)) \right) \right| \\
&\leq \sum_t^T M_0 (\frac{\varepsilon}{(T-t)2M_0} + \frac{\varepsilon}{2(T-t)M_0}),
\end{aligned}$$

which gives the result according to Definition 8.

Therefore, the existence of a solution to (Cumulative CL) depends on whether we can ensure absolute continuity of $\int_0^t \left(\int_{\mathbf{x} \in \boldsymbol{\mathcal{X}}(\tau)} \ell(\hat{f}(\mathbf{w}(t), \psi(t))(\mathbf{x})) \right) \, d\, \tau$ or not.

4.2 Reachability to a Solution in the Presence of Dissimilar Tasks

Theorem 9 shows the condition of existence of the CL solution for (Cumulative CL). The contrapositive reveals that dissimilar loss values imply dissimilar tasks. For an arbitrary CL problem, however, dissimilar tasks may or may not result in dissimilar loss values. That is, if we have two tasks $\boldsymbol{\mathcal{X}}(t)$ and $\boldsymbol{\mathcal{X}}(t + \Delta t)$ such that $\mu(\boldsymbol{\mathcal{X}}(t) \triangle \boldsymbol{\mathcal{X}}(t + \Delta t)) \geq \delta$, we do not know whether $|J(\mathbf{w}(t), \psi(t), \boldsymbol{\mathcal{X}}(t)) - J(\mathbf{w}(t+1), \psi(t+1), \boldsymbol{\mathcal{X}}(t+1))| \leq \varepsilon$ (i.e., similar loss values) or $|J(\mathbf{w}(t), \psi(t), \boldsymbol{\mathcal{X}}(t)) - J(\mathbf{w}(t+1), \psi(t+1), \boldsymbol{\mathcal{X}}(t+1))| \geq \varepsilon$ (i.e., dissimilar loss values). Nonetheless, if dissimilar tasks do produce dissimilar loss values and the number of tasks increases, the value of $|J(\mathbf{w}(t), \psi(t), \boldsymbol{\mathcal{X}}(t)) - J(\mathbf{w}(t+1), \psi(t+1), \boldsymbol{\mathcal{X}}(t+1))|$ will tend to increase and eventually exceed ε. Therefore, the model will gradually forget prior tasks, and eventually CL will fail, thus violating Theorem 9.

In order to counteract this situation, the key condition for absolute continuity must be upheld; that is, for a series of tasks, the upper bound on $\sum_t^T [|J(\mathbf{w}(\tau + \Delta\tau), \psi(\tau + \Delta\tau), \boldsymbol{\mathcal{X}}(\tau + \Delta\tau)) - J(\mathbf{w}(t), \psi(t), \boldsymbol{\mathcal{X}}(t))|]$ must be guaranteed to be less than or equal to ε. The key here is to identify the knobs in the model that can be tuned to ensure that ε is small even in the presence of dissimilar tasks. In the typical neural network learning

problem, the two knobs that can be tuned are the weights and architecture of the network. We will cement these ideas in the following and begin by elucidating the dependency of $\left(\int_{\mathbf{x}\in\boldsymbol{\mathcal{X}}(\tau)} \ell(\hat{f}(\mathbf{w}(t),\psi(t))(\mathbf{x}))\right)$ on the weights and the architecture.

Lemma 10 *Suppose $\boldsymbol{\mathcal{X}}(t)$ and $\boldsymbol{\mathcal{X}}(t+\Delta T)$ are two consecutive tasks in the measure space $(\overline{\boldsymbol{\mathcal{X}}}, \mathcal{B}(\overline{\boldsymbol{\mathcal{X}}}), \mu)$. Set M_0 to be the upper bound on the loss function ℓ across all tasks. Then for $\mu(\boldsymbol{\mathcal{X}}(t) \triangle \boldsymbol{\mathcal{X}}(t+\Delta t)) \geq \delta$, the following holds:*

$$
\begin{aligned}
E\left(\bigcup_{\tau=0}^{t+1} \boldsymbol{\mathcal{X}}(\tau)\right) = E\left(\bigcup_{\tau=0}^{t} \boldsymbol{\mathcal{X}}(\tau)\right) + \Delta t\Big[\mu\left(\bigcup_{\tau=0}^{t} \boldsymbol{\mathcal{X}}(\tau) \Delta \bigcup_{\tau=0}^{t+1} \boldsymbol{\mathcal{X}}(\tau)\right) \cdot \int_{\bigcup_{\tau=0}^{t} \boldsymbol{\mathcal{X}}(\tau) \Delta \bigcup_{\tau=0}^{t+1} \boldsymbol{\mathcal{X}}(\tau)} \ell(\hat{f}(\mathbf{w},\psi)) d\mu \\
+ \int_{\bigcup_{\tau=0}^{t} \boldsymbol{\mathcal{X}}(\tau)} \ell'(\hat{f}(\mathbf{w},\psi)) \cdot \partial^1_{\mathbf{w}} \hat{f}(\mathbf{w},\psi) \cdot \Delta w \, d\mu \\
+ \int_{\bigcup_{\tau=0}^{t} \boldsymbol{\mathcal{X}}(\tau)} \ell'(\hat{f}(\mathbf{w},\psi)) \cdot \partial^1_{\psi} \hat{f}(\mathbf{w},\psi) \cdot \Delta \psi \, d\mu\Big] + o(\Delta t). \qquad (9)
\end{aligned}
$$

Proof 4 *Note that the first-order Taylor series expansion of $E(\boldsymbol{\mathcal{X}}(t+\Delta t))$ about t is given by*

$$
\begin{aligned}
E\left(\bigcup_{\tau=0}^{t+1} \boldsymbol{\mathcal{X}}(\tau)\right) = E\left(\bigcup_{\tau=0}^{t} \boldsymbol{\mathcal{X}}(\tau)\right) + \Delta t\Big[E_{\boldsymbol{\mathcal{X}}}\left(\bigcup_{\tau=0}^{t+1} \boldsymbol{\mathcal{X}}(\tau) \Delta \bigcup_{\tau=0}^{t} \boldsymbol{\mathcal{X}}(\tau)\right) \\
+ E_{\mathbf{w}}\left(\bigcup_{\tau=0}^{t} \boldsymbol{\mathcal{X}}(\tau)\right) + E_{\psi}\left(\bigcup_{\tau=0}^{t+1} \boldsymbol{\mathcal{X}}(\tau)\right)\Big] + o(\Delta t), \qquad (10)
\end{aligned}
$$

where $E_{\boldsymbol{\mathcal{X}}}, E_{\mathbf{w}}$, and E_{ψ} are the partial derivatives of the expected value function for the data, weights, and architecture, respectively. Now, we work on acquiring each partial derivative. Toward that end,

$$
E_{\boldsymbol{\mathcal{X}}}\left(\bigcup_{\tau=0}^{t+1} \boldsymbol{\mathcal{X}}(\tau) \Delta \bigcup_{\tau=0}^{t} \boldsymbol{\mathcal{X}}(\tau)\right) = \mu\left(\left(\bigcup_{\tau=0}^{t+1} \boldsymbol{\mathcal{X}}(\tau) \Delta \bigcup_{\tau=0}^{t} \boldsymbol{\mathcal{X}}(\tau)\right)\right) \cdot \int_{\left(\bigcup_{\tau=0}^{t+1} \boldsymbol{\mathcal{X}}(\tau) \Delta \bigcup_{\tau=0}^{t} \boldsymbol{\mathcal{X}}(\tau)\right)} \ell(\hat{f}(\mathbf{w},\psi)) d\mu. \qquad (11)
$$

To determine the remaining two derivatives, we use the Sobolev function chain rule found in Evans (2022). We can do so because ℓ is real-valued and bounded and ℓ' is continuously differentiable. Then,

$$
E_{\mathbf{w}}\left(\bigcup_{\tau=0}^{t} \boldsymbol{\mathcal{X}}(\tau)\right) = \int_{\bigcup_{\tau=0}^{t} \boldsymbol{\mathcal{X}}(\tau)} \ell'(\hat{f}(\mathbf{w},\psi)) \cdot \partial^1_{\mathbf{w}} \hat{f}(\mathbf{w},\psi) \cdot \Delta w \, d\mu., \qquad (12)
$$

$$
E_{\psi}\left(\bigcup_{\tau=0}^{t} \boldsymbol{\mathcal{X}}(\tau)\right) = \int_{\bigcup_{\tau=0}^{t} \boldsymbol{\mathcal{X}}(\tau)} \ell'(\hat{f}(\mathbf{w},\psi)) \cdot \partial^1_{\psi} \hat{f}(\mathbf{w},\psi) \cdot \Delta \psi \, d\mu. \qquad (13)
$$

Substituting (11), (12), and (13) into the Taylor series expansion (10), we have

$$
\begin{aligned}
E\left(\bigcup_{\tau=0}^{t+1} \boldsymbol{\mathcal{X}}(\tau)\right) = E\left(\bigcup_{\tau=0}^{t} \boldsymbol{\mathcal{X}}(\tau)\right) + \Delta t\Big[\mu\left(\bigcup_{\tau=0}^{t} \boldsymbol{\mathcal{X}}(\tau) \Delta \bigcup_{\tau=0}^{t+1} \boldsymbol{\mathcal{X}}(\tau)\right) \cdot \int_{\bigcup_{\tau=0}^{t} \boldsymbol{\mathcal{X}}(\tau) \Delta \bigcup_{\tau=0}^{t+1} \boldsymbol{\mathcal{X}}(\tau)} \ell(\hat{f}(\mathbf{w},\psi)) d\mu \\
+ \int_{\bigcup_{\tau=0}^{t} \boldsymbol{\mathcal{X}}(\tau)} \ell'(\hat{f}(\mathbf{w},\psi)) \cdot \partial^1_{\mathbf{w}} \hat{f}(\mathbf{w},\psi) \cdot \Delta w \, d\mu \\
+ \int_{\bigcup_{\tau=0}^{t} \boldsymbol{\mathcal{X}}(\tau)} \ell'(\hat{f}(\mathbf{w},\psi)) \cdot \partial^1_{\psi} \hat{f}(\mathbf{w},\psi) \cdot \Delta \psi \, d\mu\Big] \\
+ o(\Delta t).
\end{aligned}
$$

Colloquially, in the CL problem only the weights are updated, but in this paper we argue that this is not enough. We show that with just the weights being updated, the $J(\mathbf{w}(t), \psi(t), \boldsymbol{\mathcal{X}}(t))$ eventually violates Theorem 9. To prove this, we first derive a lower bound on $\left(\int_{\mathbf{x}\in\boldsymbol{\mathcal{X}}(\tau)} \ell(\hat{f}(\mathbf{w}(t),\psi(t))(\mathbf{x}))\right)$. For the remaining analysis, we will set $\Delta t = 1$.

Lemma 11 *Suppose $\boldsymbol{\mathcal{X}}(t)$ and $\boldsymbol{\mathcal{X}}(t+\Delta t)$ are two consecutive tasks in the measure space $(\overline{\boldsymbol{\mathcal{X}}}, \mathcal{B}(\overline{\boldsymbol{\mathcal{X}}}), \mu)$. Let $L_1 > 0$ and $0 \leq \delta \leq 1$ be constants. Set M_0 to be the upper bound on the loss function ℓ across all tasks, and assume $\mu\left(\bigcup_{\tau=0}^{t} \boldsymbol{\mathcal{X}}(\tau) \Delta \bigcup_{\tau=0}^{t+\Delta t} \boldsymbol{\mathcal{X}}(\tau)\right) \geq \delta$ for all $t \in [0, T]$. Then for $\Delta t = 1$ the following holds:*

$$\sum_{\tau=t}^{T} \left(J(\boldsymbol{\mathcal{X}}(\tau)) - J(\boldsymbol{\mathcal{X}}(\tau+1))\right) \leq \sum_{\tau=t}^{T} \left(M_0 \cdot \delta - \int_{\bigcup_{\nu=0}^{\tau} \boldsymbol{\mathcal{X}}(\nu)} \left(\ell'(\hat{f}(\mathbf{w}, \psi)) \cdot \partial_{\mathbf{w}}^{1} \hat{f}(\mathbf{w}, \psi)\right)^2 \, d\mu\right)$$

for $\int_{\boldsymbol{\mathcal{X}}(t)} \ell'(\hat{f}(\mathbf{w}, \psi)) \cdot \partial_{\psi}^{1} \hat{f}(\mathbf{w}, \psi) \cdot \Delta\psi \;\; d\mu + o(\Delta t) = 0$ *as* $\psi(t) = \psi, \forall t$.

Proof 5 *The proof is in Section B.1.*

A bound similar to the one in Lemma 11 can be attained for the architecture terms as well.

Lemma 12 *Suppose $\boldsymbol{\mathcal{X}}(t)$ and $\boldsymbol{\mathcal{X}}(t+\Delta t)$ are two consecutive tasks in the measure space $(\overline{\boldsymbol{\mathcal{X}}}, \mathcal{B}(\overline{\boldsymbol{\mathcal{X}}}), \mu)$. Let $L_1 > 0$ and $0 \leq \delta \leq 1$ be constants. Set M_0 to be the upper bound on the loss function ℓ across all tasks, and assume $\mu\left(\bigcup_{\tau=0}^{t} \boldsymbol{\mathcal{X}}(\tau) \Delta \bigcup_{\tau=0}^{t+\Delta t} \boldsymbol{\mathcal{X}}(\tau)\right) \geq \delta$ for all $t \in [0, T]$. Then for $\Delta t = 1$ the following holds:*

$$\begin{aligned}\sum_{\tau=t}^{T} \left(J(\boldsymbol{\mathcal{X}}(\tau)) - J(\boldsymbol{\mathcal{X}}(\tau+1))\right) \leq \sum_{\tau=t}^{T} \Bigg(M_0 \cdot \delta &- \int_{\bigcup_{\nu=0}^{\tau} \boldsymbol{\mathcal{X}}(\nu)} \left(\ell'(\hat{f}(\mathbf{w}, \psi)) \cdot \partial_{\mathbf{w}}^{1} \hat{f}(\mathbf{w}, \psi)\right)^2 \, d\mu \\ &- \int_{\bigcup_{\nu=0}^{\tau} \boldsymbol{\mathcal{X}}(\nu)} \left(\ell'(\hat{f}(\mathbf{w}, \psi)) \cdot \partial_{\psi}^{1} \hat{f}(\mathbf{w}, \psi)\right)^2 \, d\mu\Bigg).\end{aligned}$$

Proof 6 *The proof is in Section B.2.*

The following result tells us that in the case of dissimilar tasks and with dissimilar loss values ($\sum_{\tau=t}^{T} \left(J(\boldsymbol{\mathcal{X}}(\tau)) - J(\boldsymbol{\mathcal{X}}(\tau+1))\right) \geq \varepsilon$) for a fixed architecture across tasks, we can learn the architecture at each task such that we can acquire similar loss values tasks ($\sum_{\tau=t}^{T} \left(J(\boldsymbol{\mathcal{X}}(\tau)) - J(\boldsymbol{\mathcal{X}}(\tau+1))\right) < \varepsilon$).

Theorem 13 *Let $\varepsilon > 0$, and choose δ J to be absolutely continuous. Suppose $\boldsymbol{\mathcal{X}}(t)$ and $\boldsymbol{\mathcal{X}}(t+\Delta t)$ are two consecutive tasks in the measure space $(\overline{\boldsymbol{\mathcal{X}}}, \mathcal{B}(\overline{\boldsymbol{\mathcal{X}}}), \mu)$. Set M_0 to be the upper bound on the loss function ℓ across all tasks, and assume $\mu\left(\bigcup_{\tau=0}^{t} \boldsymbol{\mathcal{X}}(\tau) \Delta \bigcup_{\tau=0}^{t+1} \boldsymbol{\mathcal{X}}(\tau)\right) \geq \delta$ for all $t \in [0, T]$. If for a fixed architecture ψ^* for all $t \in [0, T]$*

$$\sum_{\tau=t}^{T} \left(J(\mathbf{w}(\tau), \psi^*, \boldsymbol{\mathcal{X}}(\tau)) - J(\mathbf{w}(\tau+1), \psi^*, \boldsymbol{\mathcal{X}}(\tau+1))\right) \geq \varepsilon,$$

then we can choose a new architecture $\psi(t)$ for $[0, T]$ such that we can have

$$\sum_{\tau=t}^{T} \left(J(\mathbf{w}(\tau), \psi^*, \boldsymbol{\mathcal{X}}(\tau)) - J(\mathbf{w}(\tau+1), \psi(\tau), \boldsymbol{\mathcal{X}}(\tau+1))\right) < \varepsilon.$$

Proof 7 *Let $\varepsilon > 0$, and choose δ J to be absolutely continuous. As*

$$\int_{\bigcup_{\nu=0}^{\tau} \boldsymbol{\mathcal{X}}(\nu)} \left(\ell'(\hat{f}(\mathbf{w}, \psi)) \cdot \partial_{\psi}^{1} \hat{f}(\mathbf{w}, \psi)\right)^2 > 0,$$

notice that

$$\begin{aligned}\sum_{\tau=t}^{T}\left(J(\boldsymbol{\mathcal{X}}(\tau))-J(\boldsymbol{\mathcal{X}}(\tau+1))\right) &\leq \sum_{\tau=t}^{T}\Big(M_0\cdot\delta-\int_{\bigcup_{\nu=0}^{\tau}\boldsymbol{\mathcal{X}}(\nu)}\Big(\ell'(\hat{f}(\mathbf{w},\psi))\cdot\partial^1_{\mathbf{w}}\hat{f}(\mathbf{w},\psi)\Big)^2\,d\mu\\ &\quad-\int_{\bigcup_{\nu=0}^{\tau}\boldsymbol{\mathcal{X}}(\nu)}\Big(\ell'(\hat{f}(\mathbf{w},\psi))\cdot\partial^1_{\psi}\hat{f}(\mathbf{w},\psi)\Big)^2\,d\mu\Big)\\ &\leq \sum_{\tau=t}^{T}\Big(M_0\cdot\delta-\int_{\bigcup_{\nu=0}^{\tau}\boldsymbol{\mathcal{X}}(\nu)}\Big(\ell'(\hat{f}(\mathbf{w},\psi))\cdot\partial^1_{\mathbf{w}}\hat{f}(\mathbf{w},\psi)\Big)^2\,d\mu\Big).\end{aligned}$$

Thus, by changing the architecture we may be able to achieve

$$\begin{aligned}\sum_{\tau=t}^{T}\left(J(\boldsymbol{\mathcal{X}}(\tau))-J(\boldsymbol{\mathcal{X}}(\tau+1))\right) &\leq \sum_{\tau=t}^{T}\Big(M_0\cdot\delta-\int_{\bigcup_{\nu=0}^{\tau}\boldsymbol{\mathcal{X}}(\nu)}\Big(\ell'(\hat{f}(\mathbf{w},\psi))\cdot\partial^1_{\mathbf{w}}\hat{f}(\mathbf{w},\psi)\Big)^2\,d\mu\\ &\quad-\int_{\bigcup_{\nu=0}^{\tau}\boldsymbol{\mathcal{X}}(\nu)}\Big(\ell'(\hat{f}(\mathbf{w},\psi))\cdot\partial^1_{\psi}\hat{f}(\mathbf{w},\psi)\Big)^2\,d\mu\Big)\\ &< \varepsilon\\ &\leq \sum_{\tau=t}^{T}\Big(M_0\cdot\delta-\int_{\bigcup_{\nu=0}^{\tau}\boldsymbol{\mathcal{X}}(\nu)}\Big(\ell'(\hat{f}(\mathbf{w},\psi))\cdot\partial^1_{\mathbf{w}}\hat{f}(\mathbf{w},\psi)\Big)^2\,d\mu\Big).\end{aligned}$$

Thus, we have reduced the difference in loss values below the ε threshold to gain similar loss values when we began with dissimilar tasks.

The implications of Theorem 13 indicate that it is possible to shrink the effect of the new task on the forgetting cost J by changing the architecture of the network by pushing ε closer to zero. The closer ε is to zero, the more the cardinality of the intersection space and the more the task sets are connected. Therefore, upholding the absolute continuity of J is feasible.

It is now of interest to identify how to change the architecture and efficiently learn over a series of tasks. In particular, we must examine to what extent we can control the effect of architecture change on the CL problem.

5 CL Solution

To understand how to change the architecture of the model and learn over a series of tasks within the CL paradigm, we consider Figure 5. We begin at task $t = 1$ and determine an optimal weight denoted $w^*(1) = w^\infty(1)$. This gives us a neural network solution in $\mathcal{F}_1$. Then, as the next task is observed, we perform a derivative-free hyperparameter search, in particular, the directional direct search method Larson et al. (2019), to obtain a new architecture. The solution to this search is denoted $\psi^\infty(2)$ as in Figure 5. Notably, a new quandary arises since the new architecture introduces a parameter space that may have a different size from the one used for the previous task. Unfortunately, it is not feasible to trivially transfer information from the previous architecture to the new one. The common and state-of-the-art solution to this problem is to initialize the parameters in the new architecture from random and retrain on all available tasks, which is resource heavy as well as impractical. Instead, we develop a low-rank transfer algorithm that seeks to transfer information between the previous architecture to the new architecture in an efficient way. Mathematically, our goal is to solve the following bi-level problem:

$$V^*(t,\mathbf{w}(t)) = \min_{\mathbf{w}\in\mathcal{W}(\psi^*(t))} V(t,\mathbf{w}(t)), \quad \text{where } \psi^*(t) = \arg\min_{\psi\in\Psi} J(\mathbf{w}(t),\psi,\boldsymbol{\mathcal{X}}(t)). \tag{bi-level}$$

Noting that the weight search space at each t is a function of the optimal architecture, we change the weights from $t \to t + \Delta\ t$ as

$$\mathbf{w}(t+\Delta\ t) = \mathrm{A}^*(t)\mathbf{w}(t)\mathrm{B}(t)^{*T}, \tag{14}$$

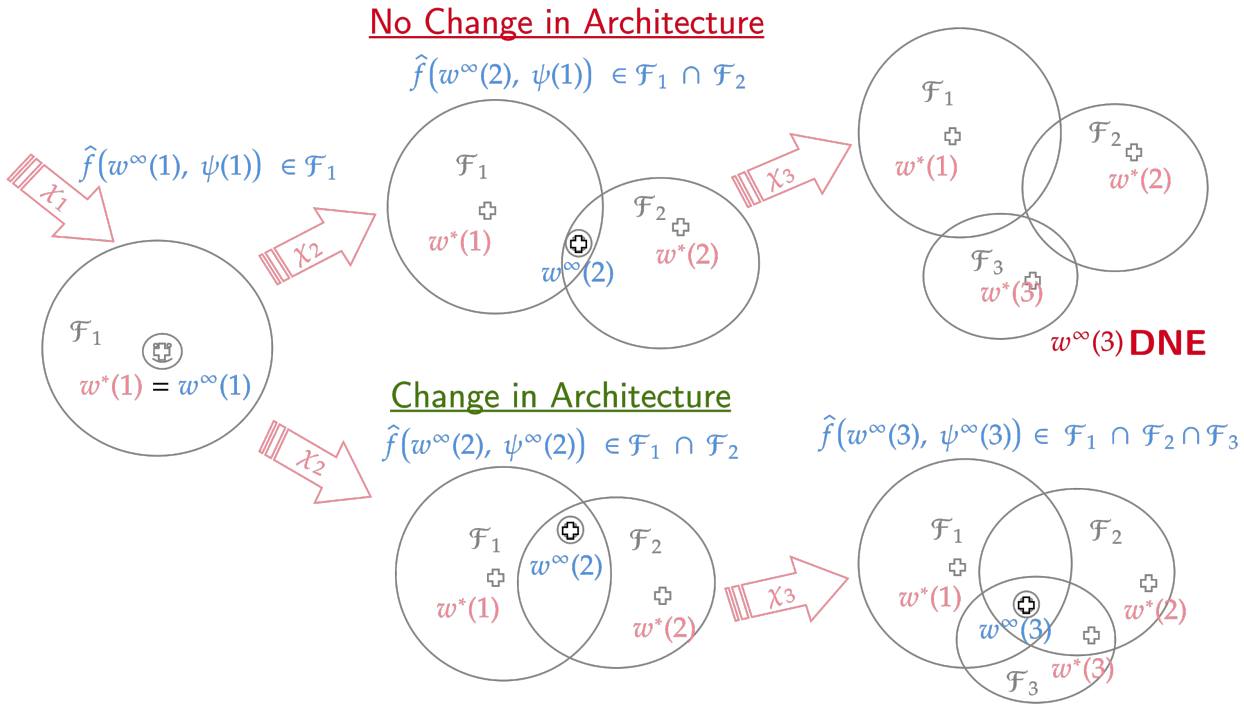


Figure 5: CL solution, where we change the size of the intersection space by introducing more capacity, through choosing novel hyperparameters

where A and B are matrices that enable the transfer of information between two different parameter spaces and where $\mathrm{A}^*, \mathrm{B}^*$ are the optimal values such that loss of information is minimal. This change in the weight space for each new task introduces dynamics into the continual learning problem defined by V^*. The following result therefore provides the total variation (the dynamics) in V^* as a function of tasks.

Proposition 14 *The total variation in $V^*(t)$ at any given task t is given by*

$$-\partial_t^1 V^*(t) = \min_{\mathbf{w} \in \mathcal{W}(\psi^*(t))} J(\mathbf{w}(t), \psi^*(t), \boldsymbol{\mathcal{X}}(t)) + \partial_{\boldsymbol{\mathcal{X}}}^1 d_t \boldsymbol{\mathcal{X}} + \partial_{\mathbf{w}}^1 V^*(t) \left[\mathrm{A}^*(t)\mathbf{w}(t)(\mathrm{B}^*(t))^T + u(t)\right], \tag{15}$$

where $\mathrm{A}^*(t), B^*(t)$ are optimal $\mathrm{A}(t)$ and $\mathrm{B}(t)$ for task t and where $u(t)$ represents the updates made to the each weights matrix of the new dimensions.

Proof 8 *The proof is in Section B.3.*

Observe that (15) is missing the term $\frac{\partial^1 V^*}{\partial^1 \psi} \frac{\partial^1 \psi}{\partial^1 t}$. This omission is intentional because the exact effects of architecture cannot be measured through derivatives in this partial differential equation (PDE). Instead, in this PDE the effect is absorbed through the change in tensors $\mathrm{A}(t)$ and $\mathrm{B}(t)$. Because of this absorption we can now define a lower bound on $\partial_t^1 V^*(t)$. This lower bound, which is essentially a lower bound on the variation, is equivalent to the lower bound on ε in Theorem 13 with the distinction that the lower bound is derived on V^* instead of just the forgetting cost J. The idea covers the effects of the dynamic program. We derive this bound below.

Theorem 15 *Given proposition 14, assume there exists a stochastic-gradient-based optimization procedure, and choose $u(t) = -\sum^I \alpha(i) \mathrm{g}^{(i)}$, where I is the number of updates and $\alpha(i)$ is some learning rate. Choose $\alpha(i)$ such that $\|\mathrm{g}_{\mathrm{MIN}}\|_{W^{k,p}(\mathrm{D})} \left\|\sum^I \alpha(i)\right\|_{W^{k,p}(\mathrm{D})} \to 0$ as $I \to \infty$ and $\min_{\mathbf{w} \in \mathcal{W}(\psi^*(t))} J(\mathbf{w}(t), \psi^*(t), \boldsymbol{\mathcal{X}}(t)) \leq \varepsilon$. Then, as long as the architecture is chosen such that $\rho_{\mathrm{MAX}}^{(\mathrm{x})} \delta =$*

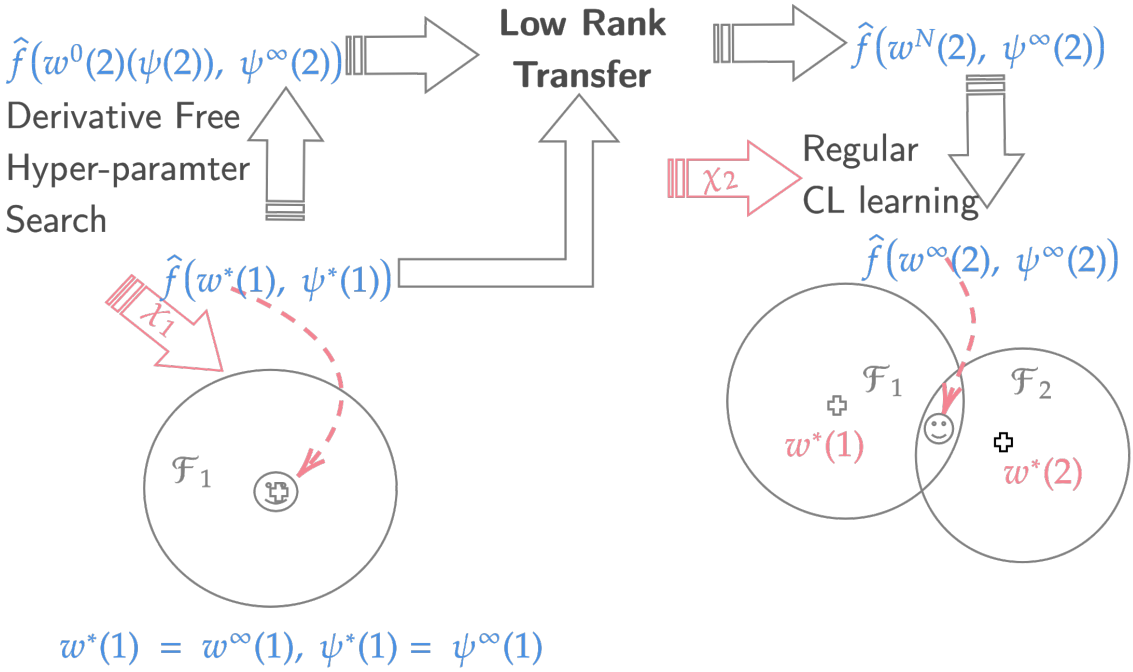


Figure 6: How we do this?

$\left[\rho^{(\mathbf{w})}_{\text{MIN}} \|\text{g}_{\text{MIN}}\|_{W^{k,p}(\text{D})} \left\|\sum^I \alpha(i)\right\|_{W^{k,p}(\text{D})} - \rho^{(\mathbf{w})}_{\text{MAX}} \left\|\text{A}^*(t)\mathbf{w}(t)(\text{B}^*(t))^T\right\|_{W^{k,p}(\text{D})}\right]$, *the total variance in* V^* *is bounded by* ε.

Proof 9 *The proof is in Section B.4.*

Therefore, even when $\alpha(i)$ is chosen such that $\|\text{g}_{\text{MIN}}\|_{W^{k,p}(\text{D})} \left\|\sum^I \alpha(i)\right\|_{W^{k,p}(\text{D})}$ as $I \to \infty$, $\rho^{(\text{x})}_{\text{MAX}}\delta$ can be countered by $\rho^{(\mathbf{w})}_{\text{MAX}} \left\|\text{A}^*(t)\mathbf{w}(t)(\text{B}^*(t))^T\right\|_{W^{k,p}(\text{D})}$, and this bound is tunable because of the change in the architecture. Therefore, as long as $\rho^{(\mathbf{w})}_{\text{MAX}} \left\|\text{A}^*(t)\mathbf{w}(t)(\text{B}^*(t))^T\right\|_{W^{k,p}(\text{D})} = \rho^{(\text{x})}_{\text{MAX}}\delta$, the total variance is bounded for all t. With the presence of bounded variance across all t, the sum of these variances is bounded. Therefore, it is clear that the V^* is absolutely continuous as long as ε is small for all $\delta > 0$.

Remark 6 *Theorem 15 is crucial because it shows that, in the presence of a perfect stochastic gradient algorithm, the change in the tasks can be countered by the choice of the architecture. By choosing a new architecture and introducing the A and B matrices such that within the CL problem the effect of varying data distribution can be mitigated, the CL problem can be acceptably solved.*

Given this guarantee, the rest of this section is dedicated to constructing an algorithm such that the condition to ensure this bounded variance is satisfied. In summary (refer to Figure 6), for every new task, we utilize an off-the-shelf algorithm to search for a new optimal architecture. Once a new architecture is found, we complete a low-rank transfer from the previous task's optimal weights to the new task parameter space. Then, using an off-the-shelf continual learning algorithm, we train the new architecture for the new task while balancing memory on all the prior tasks. In the process of low-rank transfer, we guarantee transfer of learning by requiring that the $\partial^1_t \leq \eta$, allowing for the intersection of the neural network search spaces between subsequent tasks to increase. Thus, our algorithm comprises three components: step 1 – architecture search, step 2 – low-rank transfer, and step 3 – continual learning. While step 3 is directly adapted from Chakraborty & Raghavan (2025); Raghavan & Balaprakash (2021), steps 1 and 2 are novel contributions for this work.

5.1 Step 1: Architecture Search

Although there are numerous architecture search methods that we could employ, we chose a derivative-free approach that completes a local search using finite difference approximations. This choice was made to mimic the weak derivatives available to us from viewing neural networks as functions of Sobolev spaces. As will become evident in Section 14, a notion of the change in the architecture is needed in order to model the dynamics and solve the lower optimization. We call the search method used a neighborhood directional direct-search (NDDS). The standard directional direct-search method is described in the survey in Larson et al. (2019).

The intuition behind NDDS is that we check "neighboring" architectures and compare the current architecture with the expected value of the neural network when it trains on the neighboring architecture. If the neighboring architecture has a smaller expected value, then we "move" in that direction by selecting it as the new architecture. We then check the "neighbors" of this new architecture. Since the architecture is a discrete variable, this search method provides a notion of a "gradient."

With this intuition, we now describe NDDS in depth. As we discuss this method, consult Algorithm 1. For task $t \in \boldsymbol{\mathcal{T}}$, we set $\boldsymbol{\mathcal{X}} = \mathcal{Y}$ to be our training data and $J(\mathbf{w}(t), \psi(t), \mathcal{Y})$ the expected value function. We set $x_s = \psi(t)$, and let D_s be a finite set of directions. Further, we choose a "step size" $\alpha_s \in \mathbb{N}$. Now we generate the "neighboring" points for x_s via D_s and α_s. We call these points *poll points* and define them as follows:

$$PollPoints = \{x_s + \alpha_s d_i : d_i \in D_s\}.$$

Using a randomly selected subset of our data, denoted $\mathcal{Y}_s$, we determine and compare the expected value of the neural network with randomly initialized weights according to each poll point architecture. If one of these poll points results in a smaller expected value than x_s, we set x_s equal to the poll point. We then repeat this process starting with this new architecture. This search terminates when the expected value is beneath a previously chosen threshold or after five times of repeating. In practice, we chose the threshold to be a percentage lower than the expected value produced by the original architecture.

Let us walk through an example. Suppose we would like to learn the optimal number of neurons per layer in our hidden layers for training a feedforward neural network (FNN) on a dataset. Because of the data, suppose the input layer is fixed at 784 neurons and the output layer is fixed at 10 neurons per layer. Let us fix the number of hidden layers at 2 beginning with 50 neurons in each hidden layer. We set $\mathcal{Y}_s$ to be a randomly chosen appropriately sized subset of task t data and previous task data. Our direction set is $D_s = D = \{[0, 0, 10, 0], [0, 10, 0, 0]\}$ and will be the same for every s. We start our search by setting the step size $\alpha_s = 10$. For the first round, we have $x_s = [784, 50, 50, 10]$, so our poll points are $[784, 60, 50, 10]$, $[784, 50, 60, 10]$, and $[784, 60, 60, 10]$. We then train an FNN of each size on the dataset Y_s and determine the corresponding expected values. If the expected value of the FNN $[784, 60, 50, 10]$ is smaller than that of the FNN $[784, 50, 50, 10]$, for example, then $x_s = [784, 60, 50, 10]$. We would then continue this process until the expected value of our network was less than our set threshold or we exhausted our neighborhood search.

5.2 Step 2: Low-Rank Transfer

Now that the optimal architecture (i.e., number of neurons per layer) has been determined for the current task, it is immediately obvious that the size of the weights tensor will not match with the new architecture. Thus, we seek a method to determine a weights tensor corresponding to the new optimal architecture that retains previously learned information and transfers it. We propose a method we call low-rank transfer.

Before we dive into the steps of this method, let us set some assumptions and notation. Recall that the only component of the architecture we seek to learn is the number of neurons per layer in our neural network. Thus, we assume our network has d layers, and we fix all other architecture parameters except for the number of neurons in our hidden layers. From Definition 3, $\psi_i(t)$ provides the dimensions for the corresponding weights matrix in each layer of our network.

In order of appearance, we assign the values of $\psi(t)$ to the values r_i, s_i for each i such that $1 \leq i \leq d$. If $\psi^*(t)$ represents the optimal architecture returned from the NDDS completed for task t, then similarly

Algorithm 1: Neighborhood Directional Direct Search (NDDS)

GAIL - Please decide here and throughout whether you are talking about a neighborhood direct-directional search, a direct-directional neighborhood search, or neighborhood directional direct search Input: Initial architecture x_s, step size $\alpha_0 \in \mathbb{N}$

Input: Threshold parameter threshold $\in \mathbb{R}$

Output: Updated architecture x_s

$j \leftarrow 0$;

loss $\leftarrow$ `training_loop`$(x_s, \mathcal{Y}_s)$;

while loss $>$ threshold ***or*** $j < 5$ **do**

  pollPoints $\leftarrow \{x_s + \alpha_s d_i : d_i \in D_s\}$;

  foreach poll $\in$ pollPoints **do**

    loss$_s \leftarrow$ `training_loop`$(\text{poll}, \mathcal{Y}_s)$;

    if loss $\leq$ loss$_s$ **then**

      $x_s \leftarrow x_s$;

    else

      $x_s \leftarrow$ poll;

      loss $\leftarrow$ loss$_s$;

    end

  end

  $j \leftarrow j + 1$;

end

we can assign the values of the dimensions of the weights matrices of each layer of $\psi_i^*(t)$ to be a_i, b_i for $1 \leq i \leq d$. Our goal is to utilize the original weights matrices to project into the weights matrix space for the new architecture.

To begin, we initialize dimension 3 tensors $\mathrm{A}(t), \mathrm{B}(t)$, which each comprise d matrices. Let each matrix $\mathrm{A}_i(t)$ in $\mathrm{A}(t)$ be randomly generated with dimensions $a_i \times r_i$ for $1 \leq i \leq d$. Similarly, let each matrix B_i in $\mathrm{B}(t)$ be randomly generated with size $b_i \times s_i$ for $1 \leq i \leq d$. Then, set $C(t) = \mathrm{A}(t)\mathbf{w}(t)\mathrm{B}^T(t)$. More specifically $C(t)$ comprises d matrices such that $C_i(t) = \mathrm{A}_i(t)\mathbf{w}_i(t)B_i^T(t)$ for all $1 \leq i \leq d$. Notice that the dimensions of each $C_i(t)$ are $a_i \times b_i$. These are the dimensions of the weights matrices for a neural netowrk with the new optimal architecture $\psi^*(t)$. See Figure 7 to understand the construction of $C(t)$ more explicitly.

To prevent loss of information from $\mathbf{w}(t)$, when we make the transfer to $C(t)$, we train only the $\mathrm{A}(t)$ and $\mathrm{B}(t)$ portions of the new weights tensor $C(t)$ on the task data for a chosen number of epochs, while freezing the $\mathbf{w}(t)$ tensor. This additional training ensures a transfer of learning to the new weights corresponding to the new optimal architecture. If $C^*(t)$ represents $C(t)$ after this training on just tensors $\mathrm{A}(t)$ and $\mathrm{B}(t)$ and $\psi(t+1) = \psi^*(t)$, then we conclude our process by letting $w(t+1) = C^*(t)$ and completing the standard training of our neural network $\hat{f}(w(t+1), \psi(t+1))$ on task data. In Algorithm 2 we summarize each step of our method described in the preceding subsections.

6 Related Works

As noted in the introduction, theoretical and empirical results have proven that even in traditional machine learning, training weights of the network alone is not enough. By utilizing tools such as NAS (Neural Architecture Search), LoRA (Low-Rank Adaptation), and PEFT (Parameter-Efficient Fine Tuning), the accuracy of a trained model can readily be increased (Liu et al. (2021), Hu et al. (2022), Han et al. (2024)). In the context of CL, however, learning optimal hyperparameters or architecture for each task poses a challenge: learning of previous tasks be transferred.

Toward this end, networks with dynamic structures, such as progressive neural networks (PNGs)

and dynamically expandable networks (DENs) were introduced in Rusu et al. (2016) and Yoon et al. (2017). **GAIL - do you mean progressive generative networks (PGNs) or progressive neural networks**

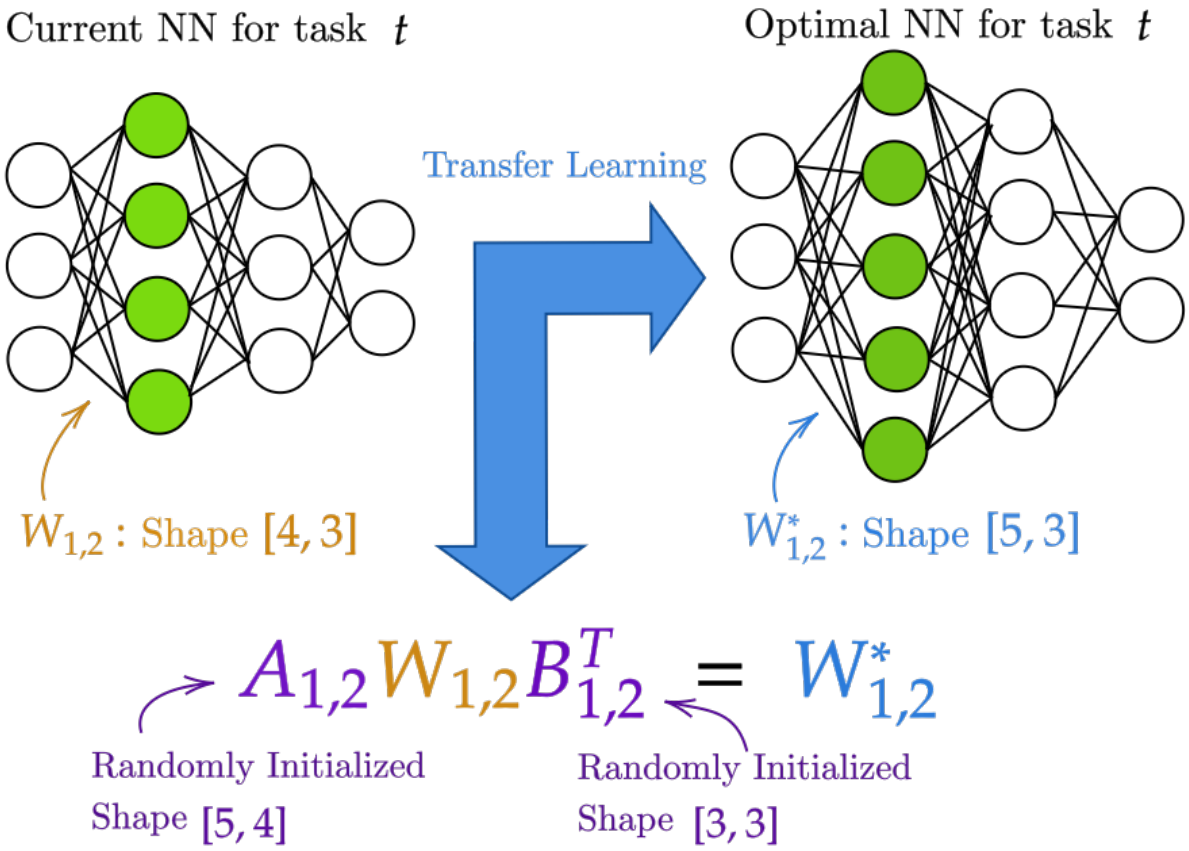


Figure 7: Method of low-rank transfer at a single layer

Algorithm 2: Main Training Loop

Choose $\mathbf{w}(t)$ and $\psi(t)$ to begin
Set `epoch hyper-parameter`
for $t = 0, 1, \ldots, T$ **do**
  Step 1: Standard Training of $\mathbf{w}(t)$
  $\mathbf{w}(t) \leftarrow$`training_loop`$(\mathbf{w}(t), \psi(t),\ \boldsymbol{\mathcal{X}}(t),$ `epochs`$)$
  Step 2: Architecture Search
  $\psi^*(t) \leftarrow$ `NDDS`$(\psi(t),\ \boldsymbol{\mathcal{X}}(t))$
  Step 3: Initialize $\mathrm{A}(t), \mathrm{B}(t)$
  for $i = 1, ..., d$ **do**
    $\mathrm{A}_i(t), \mathrm{B}_i \leftarrow$`init_AB`$(a_i, b_i, r_i, s_i)$
  end
  Step 4: Set $C(t)$
  for $i = 1, ..., d$ **do**
    $C_i(t) = \mathrm{A}_i(t)\mathbf{w}_i(t)B_i^T(t)$
  end
  Step 5: Fix $\mathbf{w}(t)$, Train $\mathrm{A}(t), \mathrm{B}(t)$ for $\hat{f}(C(t), \psi(t+1))$
  $C^*(t) \leftarrow$ `train_AB`$(C(t), \psi(t+1),\ \boldsymbol{\mathcal{X}}(t),$ `epochs`$)$
  Step 6: Set New Weights and Architecture
  $w(t+1) = C^*(t)$
  $\psi(t+1) = \psi^*(t)$
  Step 7: Standard Training on New NN `training_loop`
  $(\mathbf{w}(t+1), \psi(t+1),\ \boldsymbol{\mathcal{X}}(t),$ `epochs`$)$
end

)PNNs) ? I kept to PNN since the reference is to that, buy you need to check here) PNNs add a layer to the neural network at each observed task Rusu et al. (2016). While the model's accuracy did improve with such structure, extensive compute time is unavoidable as more tasks, and hence layers, are observed. DENs attempted to remedy the computing issues of PNNs. DENs first determine a subcollection of neurons to train on for a given task (via a metric) and then add a layer of neurons to the network where appropriate Yoon et al. (2017). This method also completes sparse regularization at each task to prune the network in an attempt to reduce compute time compared with PNNs. The use of a DEN did improve the model's accuracy. However, drawbacks such as compute time, overfitting, and transfer of learning still remained. Learning from previous tasks was not fully transferred, since the weights corresponding to the newly added network layers were randomly initialized.

In more recent years researchers have built on the idea of DENs by developing methods that optimally choose how and when to adjust network architecture. CLEAS (Continual Learning with Efficient Architecture Search) is the first integration of NAS into the CL framework Gao et al. (2022). This method changes architecture only when deemed necessary by utilizing a neuron-level NAS, rather than adding an entire layer to the neural network. Moreover, the network is pruned at each step to remove unnecessary layers and/or neurons. The algorithm touts its smaller, purposely learned network structure, which allows for reduced network complexity and hence reduced computation time. In particular, CLEAS improves model accuracy by up to 6.70% and reduces network complexity by up to 51.0% on the three benchmark datasets. A strategy similar to CLEAS is CAS (Continual Architecture Search) Pasunuru & Bansal (2019). The premise of adding and removing to the architecture only when deemed optimal for a given task remains the same. Rather than using NAS, however, they use what they call Efficient Neural Architecture Search, which incorporates a weight-sharing strategy. SEAL (Searching Expandable Architectures for Incremental Learning) again seeks to change the hyperparameters and architecture of the neural network only when necessary, but the method differs from CLEAS in how it determines when to alter the network Gambella et al. (2025). In particular, SEAL uses a capacity estimation metric to make such a decision.

The previous methods all sought to expand the network in some fashion to increase the accuracy of the network. Now we shift to how parameter-efficient fine-tuning methods have been utilized in CL. In particular, we investigate the use of LoRA (Low-Rank Adaptation) Hu et al. (2022). The two methods we will discuss here are intended for pretrained transformers that are now attempting to learn tasks. The CoLoRA method utilizes a new LoRA adapter for each new task to train a task expert model Wistuba et al. (2023). The expert model is then used to train the transformer. At the time, this method produced state-of-the-art results, but at a high computational cost. Since then, the CLoRA method has been introduced Muralidhara et al. (2025). The key difference between these two strategies is the number of LoRA adapters. Rather than reinitializing and training a new LoRA adapter for each task, a single-adapter approach is used. This reduces the compute time, while maintaining accuracy and transferring learning.

In this paper we attempt to combine the most profitable aspects of methods such as CLEAS and CLoRA. In other words, our goal is to learn the optimal architecture for each task and utilize a low-rank transfer method to optimally transfer learning to the new architecture. The use of a LoRA-like method to easily guarantee transfer of learning from one architecture to another in standard CL has not been accomplished. Moreover, no previous algorithm has provided a theoretical mathematics framework to support the empirical data.

7 Experimental Results

In this section we evaluate the algorithm in 2 and seek to answer the following questions: Does training with weights for a large number of tasks lead to a saturation as discussed in Section 2? Does changing the architecture actually helps with learning over consecutive tasks (Section B.1)? Does changing the architecture help improve saturation over a series of tasks? And do these ideas carry forward to different types of neural network architectures?

7.1 Datasets and Metrics:

We examine these questions for three continual learning problems: regression, image classification, and graph classification.

Datasets: For the regression problem, we consider the randomly generated sine dataset. For the image classification problem, we consider the Omniglot dataset, which consists of handwritten characters. For the graph classification problem, we consider a set of randomly generated graphs utilizing the FakeDataset class from the PyTorch Geometric library. We introduce additional details about these datasets in their respective sections.

Metrics: For all the experiments, we introduce a new task every 500 epochs. For each such update, we record training and test loss values. For the regression problem we use the mean squared error loss function, and for the classification problems we use the cross-entropy loss function.

7.2 Implementation Details

Experimental Infrastructure: All experiments were conducted locally on a 14-inch MacBook Pro (2024) equipped with an Apple M4 Pro system-on-chip, featuring a 14-core CPU, 20-core GPU, 16-core Neural Engine, 24 GB of unified memory, and 512 GB of SSD storage. The system ran macOS (Sequoia 15.6.1) and utilized Metal Performance Shaders (MPS) for hardware acceleration on Apple Silicon. The experimental codebase was implemented in Python using multiple machine learning frameworks, including JAX frostig et al. (2018) and the Equinox library Kidger & Garcia (2021). All training and evaluation scripts were executed directly through the macOS terminal. Experiment outputs, including learning curves, losses, and metrics, were logged and visualized by using TensorBoard. Environment isolation and package management were handled via conda/miniforge, and all runs were performed with fixed random seeds to ensure reproducibility. The code is publicly available on git at `https://github.com/krm9c/ContLearn.git`.

7.2.1 Step 1: Baseline Continual Learning Approach

While our approach is generic and can be used in conjunction with any available continual learning algorithm, we focus on baseline CL approaches and demonstrate viability with respect to different neural network architectures in this work. For our baseline comparison we use a replay-driven continual learning approach that seeks to balance forgetting and generalization demonstrated in Raghavan & Balaprakash (2021) for image classification and regression experiments. We use a replay-driven continual graph learning approach outlined in Raghavan & Balaprakash (2023) for all comparisons. We use the replay buffer size of $1,000$ for each of these comparisons.

7.2.2 Step 2: Architecture Search:

To optimize the network architectures, we employed the direct-directional search method, a derivative-free approach that iteratively explores candidate architectures using only performance evaluations. The search focused on the number of neurons in each hidden layer, while keeping the number of layers and all other architectural components fixed.

Feedforward Neural Network: For feedforward networks, the search begins with the current architecture and generates candidate architectures by incrementally adding neurons to each hidden layer. For example, given an initial architecture $[10, 40, 40, 10]$ and a step size of 5, the first candidate architectures evaluated are $[10, 45, 40, 10]$, $[10, 40, 45, 10]$, and $[10, 45, 45, 10]$. Each candidate is trained for a fixed number of epochs (e.g., 100), and the architecture that achieves the lowest training loss is selected as the new architecture for the next round. This process is repeated for a specified number of rounds. While effective at optimizing layer widths, this approach is computationally intensive, since each round requires full training of multiple candidate architectures.

Convolutional/Graph Convolutional Neural Network: For CNNs and GCNs, the search was extended to jointly optimize the number of neurons in the feedforward layers and the filter size of the convolutional or graph convolutional layers. The procedure follows a nested loop structure: for each candidate filter size (e.g., 2, 3, or 4), multiple rounds of neuron-width optimization are performed using the same incremental step-size procedure as discussed earlier. Starting from the current architecture, candidate architectures are generated by adding neurons to each hidden layer in the feedforward network blocks in the CNN and GCN. As in the feedforward neural network, each candidate is trained for a fixed number of epochs to evaluate performance. The architecture achieving the lowest training loss is selected for the next round, and the process is repeated for all considered filter sizes. The final architecture is the combination of filter size and layer widths that minimizes the loss. This double-loop procedure is also computationally expensive, because of the need to train multiple candidates for each filter size in every round.

7.2.3 Step 3: Knowledge transfer

A key component of Algorithm 2 is the need to transfer learning from the $\mathbf{w}(t) \rightarrow \mathbf{w}(t+1)$ through updates of A and B matrices. Since the size and shape of $\mathbf{w}, A, B$ change with change in the architecture, the implementation differs across different architectures. We realize our algorithm for three different architectures—a feedforward network, graph neural network, and convolutional neural network—by implementing additional methods in the neural network class and manipulating the weights of the model on the fly.

To adapt the network to the case of evolving weights, we add an additional method into the model that calculates the output of each layer when the weights of that layer are set to $A\mathbf{w}B^T$. instead of just $\mathbf{w}$. At the onset of each new architecture change, we use this method to provide the forward pass of the neural network. Toward this end, we set $\mathbf{w}$ to be nontrainable and set A and B to be trainable. Once the new architecture is generated, we update A and B through an Adam optimizer (initialized from scratch for the inner loop: step 3 in Algorithm 2) for a predetermined number of iterations (steps 4 and 5 in Algorithm 2) and then set $\mathbf{w} = A\mathbf{w}B^T$ (step 6 in Algorithm 2) to adapt the model to the new task (steps 4 and 5 in Algorithm 2).

For the convolutional network, there are two sets of A and B matrices. The first group pertain to the convolutional layers, where we change the filter-size and $\mathbf{w}$ is the filter. On the other hand, for the feedforward layers, A and B correspond to standard weight matrices. In the graph classification problem, these matrices

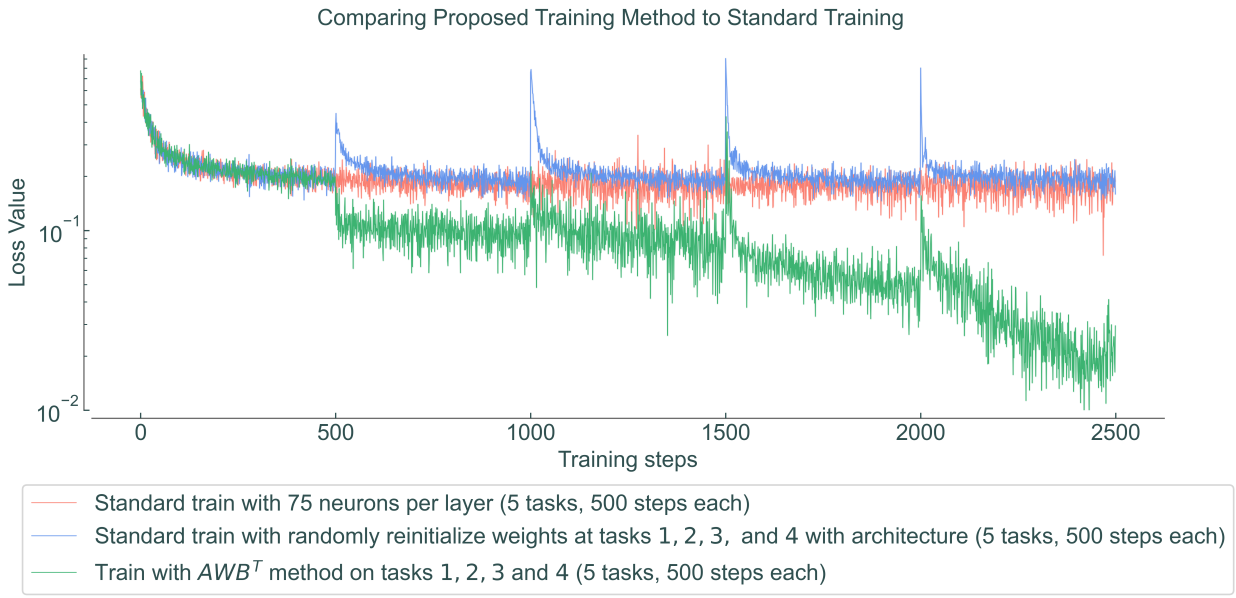


Figure 8: Experiment 1: Comparing loss values on training data for baseline continual learning method and method of learning optimal task architecture

exist for each graph convolutional layer as well as the feedforward layers, resulting in multiple distinct auxiliary matrices. In the case of graph convolution layers, A and B matrices pertain to the graph convolution filters and, for the feedforward layer, the standard weight matrices.

7.3 Regression

Dataset and Task Splits: The sine dataset is a standard benchmark for continual learning in regression settings. Each task is defined by a sine function with a distinct amplitude and/or phase, where inputs x, are sampled from a fixed domain $x \sim \mathcal{U}([-90, 90])$. This controlled variation across tasks provides a simple yet nontrivial setting to evaluate a model's ability to learn sequentially and resist catastrophic forgetting. Task division is achieved by systematically varying the amplitude and/or phase for each task, creating functionally distinct mappings that require continual adaptation.

Hyperparameter Choices: For the regression problem, we utilize a standard feedforward neural network with four layers, where we begin with 75 neurons in each of the hidden layers at the first task. We use AdamW Loshchilov et al. (2017) as the optimizer with a learning rate of 1×10^{-4} in the training regime and use Glorot uniform Glorot & Bengio (2010) to initialize the weights. We conduct three sets of experiments.

Experiment 1: For the first experiment, we learn five tasks and train 500 epochs on each task. We record in Figure 8 the loss function progression with respect to training epochs. In this experiment we first examine the behavior of a standard CL regime where no change in the architecture is performed. This serves as our baseline and is depicted by the red graph. Next, the blue graph describes the loss value if we change the architecture at each task with random re-initialization. The green graph shows the loss value when our method of changing the architecture is used at each step low-rank transfer is utilized to retain learning.

We note from the red graph that, with the introduction of new tasks, the model's loss function does not reduce, indicating a stagnation of learning. This behavior suggests that, with changing tasks, the change in the weight parameters has no effect on the training and the learning stagnates. Notably, we argued in our theory, particularly in Lemma 11, that the change in the tasks can lead to a loss of continuity of the

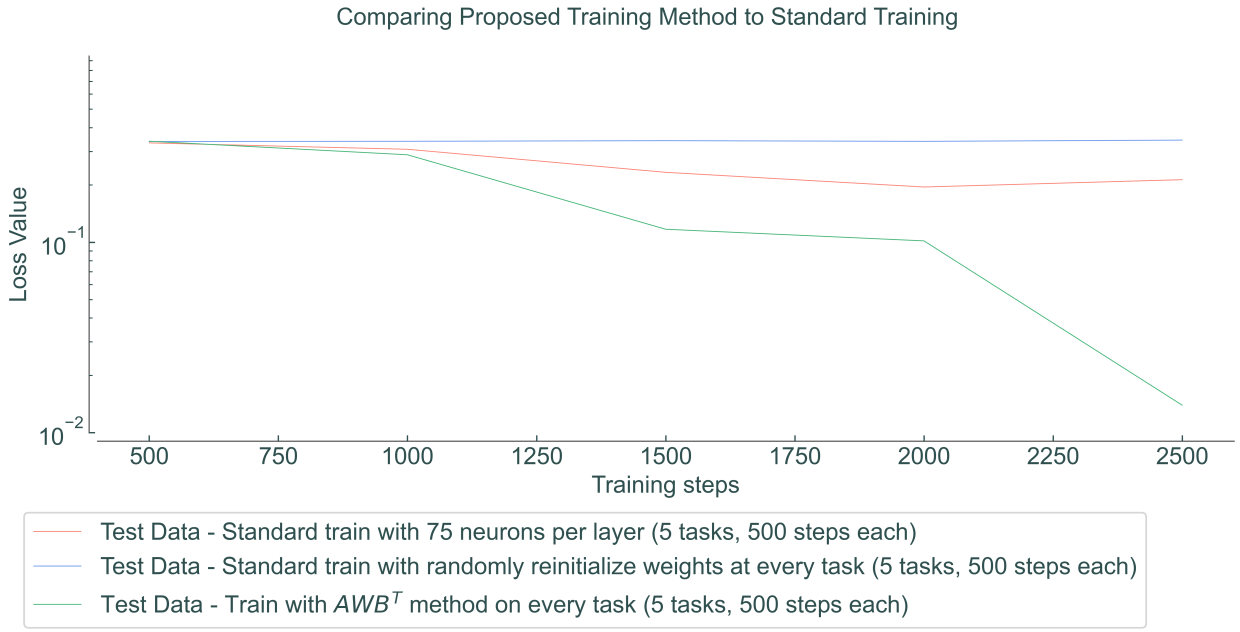


Figure 9: Experiment 1: Comparing loss values on test data for baseline continual learning method and method of learning optimal task architecture

forgetting cost, leading to the conclusion that just changing the weights is not enough. This behavior is observed through the red graph.

To obviate this behavior, we seek to change the architecture on the fly. The progression of the loss function with changing architecture on the fly is shown in the blue curve. However, this process introduces a small spike at the onset of each new task. Notably, with the blue curve the weights are initialized at random after the new architecture is chosen, and therefore the continuity of the loss function across two subsequent task is not upheld by the model, indicated by the spikes in the loss progression. We note that even with the new architecture, the model does improve over the red curve. There are two reasons. First, in many cases the initialization point of the model for the new task biases the learning for the new task, and new information therefore is not learned effectively. Second, we simply needed to learn the new task for a larger number of epochs. The blue graph suggests that changing architecture without transfer of learning from previous tasks to the next task leads to loss of learning.

The green graph displays a decrease in the loss value on the training data in comparison with the baseline continual learning and the case when the architecture is changed without transfer. We can deduce that changing the architecture and transferring learning across tasks leads to increased performance. We see that at the onset of each new task there is still a small spike in the loss function. This spike is expected because there is a change in the size of the parameter space. This implies that with the presence of transfer between the parameters spaces, the loss function drops drastically. We note that the transfer between parameter spaces not only counters the effect of changing tasks, as proved in Theorem 15, but also intuitively provides a better starting point for learning the new tasks. The difference is that learning behavior between the red curve and the green curve implies that the conclusions from Lemmas 12 and 11 are indeed valid where change in the architecture lowers in the upper bound indicated by the lower loss values. All conclusions from Figure 8 extend to Figure 9, where the test loss values are graphed.

Experiment 2: For the second experiment, we seek to determine whether the performance from the previous experiment extends to a larger number of task. In this process we learn 10 tasks of data and train 500 epochs on each task and plot the loss values in Figure 10. We compare the performance of the standard continual learning method used in Raghavan & Balaprakash (2021) with no change made to the architecture across

Figure 10: Regression Experiment 2: Comparing loss values on training data for baseline continual learning method and method of learning optimal task architecture

tasks and with our approach (represented by the blue curve) where the architecture is modified and the learning is transferred between the two tasks. We observe similar conclusions from Figures 8 and 9 where changing the architecture and transferring learning are helpful. In particular, the red curve is saturated after the first task and does not provide any improvement while the blue curve keeps on improving. After Task 6, however, no more reduction in loss is seen. It is expected that after a few iterations of architecture search and transfer of learning, a new architecture with better performance is not found. Therefore, the approach does not provide any additional improvement. This behavior is an artifact of the architecture search method where better architectures cannot be found. We noted that any architecture search approach can be introduced here, including popular methods such as DeepHyper Balaprakash et al. (2018). However, the theoretical conclusions are still valid with this approach, and the conclusions extend to the test dataset as in Figure 11.

Experiment 3: For the third experiment, we learn 5 tasks of data and train 500 epochs on each task. To understand the implications of Lemma 12 and 11, we introduce noise into each subsequent task and observe the learning behavior. Figure 12 reveals the loss values as the neural network training (on the training data). The red graph is again the standard continual learning method used in Raghavan & Balaprakash (2021), where no change is made to the architecture across tasks. Supporting the conclusions from Theorem 9 and Lemmas 12 and 11, we observe that with increasing noise there is an increase in the difference in the performance between two subsequent tasks. In other words, in the presence of changing tasks, the capacity diverges as in Theorem 1, and changing weights is not enough to maintain performance as in Lemma 11. The blue graph shows that even in the presence of noise, there is improvement in the performance of the model when we change the architecture at each step and we complete the low-rank transfer to retain learning. Since the blue graph displays a decrease in the loss value on the training data in comparison with the baseline continual learning, we conclude that changing the architecture while also transferring learning at each task increases performance even when subsequent tasks introduce large jumps in the loss function, thus violating Theorem 9.

7.4 Classification

Dataset and Task Splits: The Omniglot dataset Lake et al. (2015) is a cornerstone benchmark for few-shot and meta-learning, designed to test how well models can learn new concepts from extremely limited

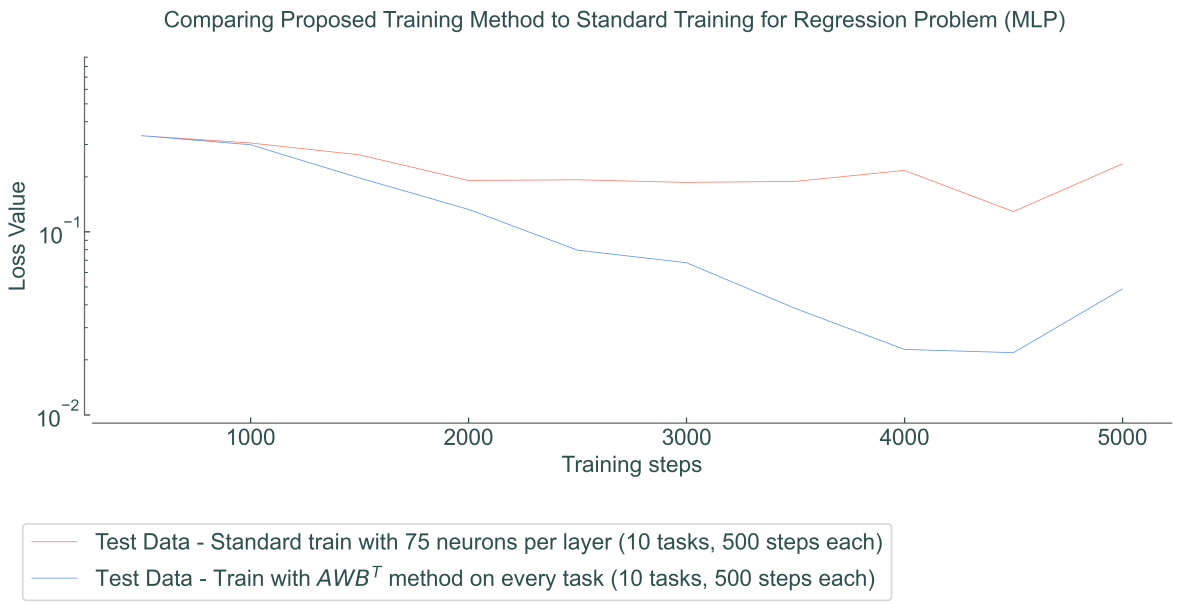


Figure 11: Regression Experiment 2: Comparing loss values on test data for baseline continual learning method and method of learning optimal task architecture

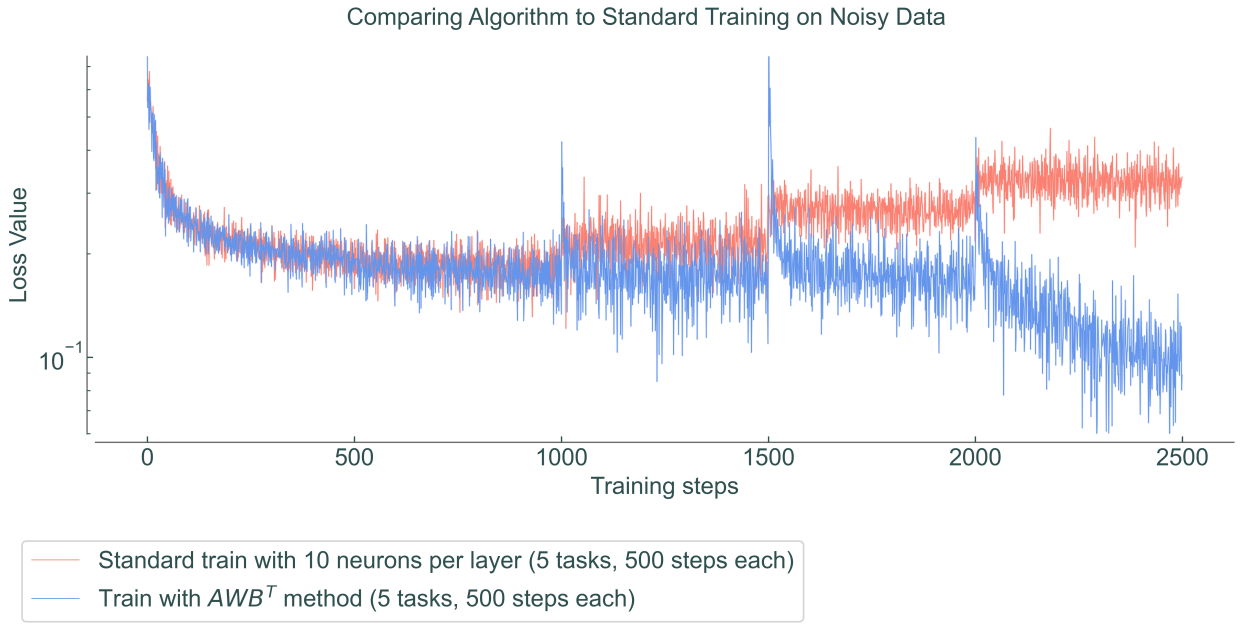


Figure 12: Regression Experiment 3: Comparing loss values on training data for baseline continual learning method and method of learning optimal task architecture when noise in the data is increased

data. Comprising $1,623$ handwritten characters from 50 diverse alphabets, each rendered 20 times by different individuals, it captures rich variation in writing style and structure. Its 105×105 grayscale images provide clean, simple inputs while still offering enough complexity to challenge learning algorithms. We split the Omniglot dataset into multiple tasks by randomly selecting three classes and applying rotational transformations on them to generate a task.

Hyperparameter Choices: For the classification experiment, we use a convolutional neural network with one feedforward layer. The convolutional layers are followed by max-pooling and a *ReLU* activation function, which extract spatial features from the input. Flattened convolutional outputs are passed through a small feedforward network with fully connected layers. We initialize the architecture with a convolutional filter size of 3 and hidden layers sizes of 512 and 64. We learn 5 tasks of data based on the Omniglot dataset and train 500 epochs on each task. We use AdamW Loshchilov et al. (2017) with a learning rate of 1×10^{-4} in the training regime.

Results: Figure 13 shows the loss function values as the CNN trains (on the training data) with respect to different methods as discussed in Section 7.3 on the five tasks. The red graph is the standard continual learning method used in Raghavan & Balaprakash (2021), where no change is made to the architecture across tasks; this serves as our baseline for comparison. The blue graph shows the progression of loss values when our proposed method is used.

For the regression experiment, we learned the number of neurons per layer through our architecture search. With this experiment, however, we determine the optimal filter size for the convolutional layers and the optimal number of neurons in the feedforward neural network. Thus, the low-rank transfer is performed for the filters, and the weights matrices are determined by the feedforward neural network. In particular, this highlights the flexibility of our method, where we can perform transfer of information between convolutional filters as well as standard weight matrices.

With the first 500 epochs of training, we observe from Figure 13 that both the blue curve and the red curve coincide. After the first task is introduced, however, the red curve has a large jump. This jump is common in classification problem because, unlike regression, classification problems do not have task-to-task dependency: each task is sampled independently, by virtue of iid sampling. This leads to a lot of jumps in the loss function and its gradients and incurs large forgetting. We note, however, that our approach counters this jump significantly with increased performance for each subsequent task. Notably, the jump in some tasks is larger compared with other tasks. We note that some of the jumps pertain to how well A and B matrices transfer information between tasks.

It was shown in Theorem 15 that the eigenvalues of the Jacobian induced by the change in the data must be countered by the singular values of A and B matrices. However, this is hard to ensure in practice, especially when the tasks are independently sampled as in the Omniglot dataset case. Although it is feasible to devise a method that exactly establishes the conditions outlined in Theorem 15, such an approach requires careful engineering and has been relegated to future work. Despite this, the conclusions from Section 7.3 carry forward to this case with the key difference that the jumps between tasks are large.

7.5 Graph Classification

Hyperparameter Choices: For the graph classification task, a graph convolutional network (GCN) was implemented that involved multiple graph convolutional layers, each transforming node features while incorporating neighborhood information via adjacency matrices. Following the convolutional layers, a graph pooling layer aggregates node-level embeddings into a fixed-size graph representation. The pooled embeddings are then processed through fully connected feedforward layers to produce graph-level predictions. The network employs leaky ReLU activations and optional dropout for regularization. We begin with a single GCN layer with filter size 3 followed by a feedforward neural network with 4 layers where the hidden layers are of size 140. We learn five tasks of data and train 125 epochs on each task. We use AdamW with a learning rate of 1×10^{-4} in the training regime. For each GCN layer, we observe the best filter size with the architecture search approach presented here.

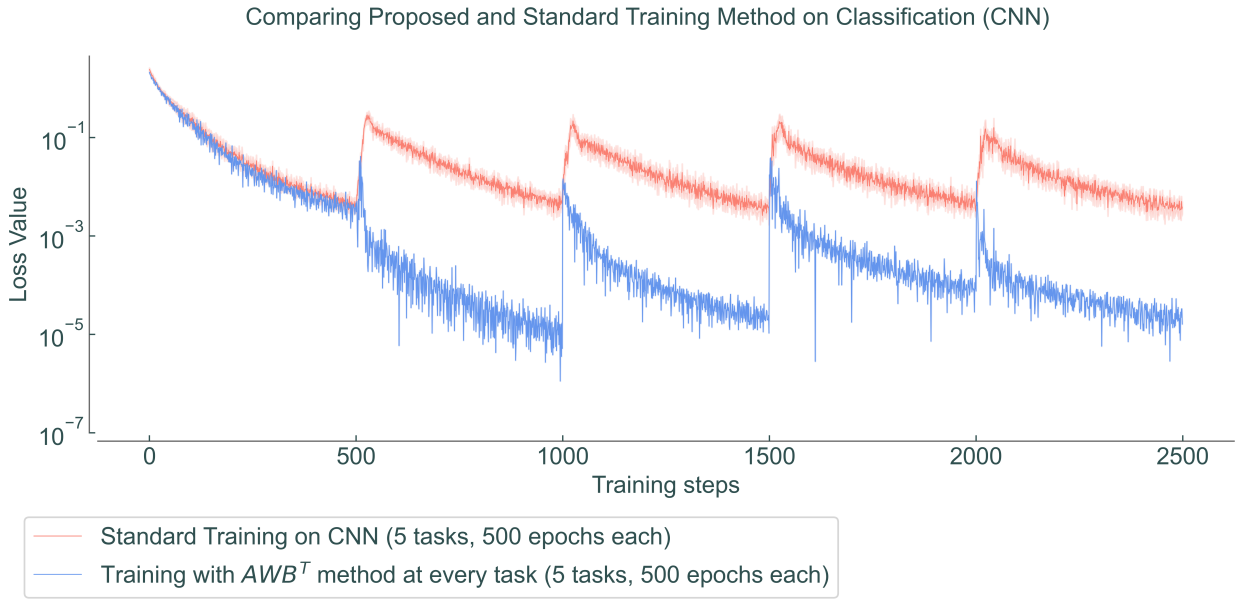


Figure 13: Classification experiment: Comparing loss values on training data for baseline continual learning method and method of learning optimal task architecture

Datasets and Task Splits: We generate our tasks by sampling through a FakeHeteroDataset class, which is a synthetic dataset class in the PyTorch Geometric Fey et al. (2025) library that generates random heterogeneous graphs. Heterogeneous graphs imply that subsequent tasks do not have the same number of nodes or edges. The datasets returned are purely random—node and edge features, connectivity, and labels.

Results: Figure 14 shows the loss values as the graph neural network trains (on the training data) with respect to different methods on the five tasks. The red graph is a graph continual learning method from Raghavan & Balaprakash (2023), equipped to consider the dynamic changes between nodes and edges in the continual learning setting. This serves as our baseline for comparison. The blue graph describes the loss value if we change the architecture at each task; but rather than utilizing the low-rank transfer (which we describe in 2), we randomly reinitialize the weights. The green graph shows the loss value with our proposed method of changing the architecture at each step and completing the low-rank transfer to retain learning.

As shown from Figure 14, the red graph stagnates in its learning, and the architecture change along with low-rank transfer allows for significant improvements in the loss function. We observe that the loss function improvements are better for graph classification than the other two cases presented in the results section. We again observe that the conclusions from image classification and regression are upheld in Figure 14, where changing the weights is not enough.

8 Conclusion

In this paper we elucidated the dependency between the architecture, weights, and loss function through a Sobolev space design. We derive the necessary condition for the existence of the continual learning problem and show that it requires the forgetting cost to be absolutely continuous. We show both theoretically and with experimentation that just changing the weights of the network is not enough to satisfy the necessary condition, and we devise an approach to change the architecture of the model on the fly. We show that our approach substantially improvemes the training and test performance even in the presence of noise in the tasks. We also show that, for a series of neural network architectures, we have better performance when changing architectures while learning continually on the series of tasks.

GAIL - throughout, are you talking about necessary condition or conditions?

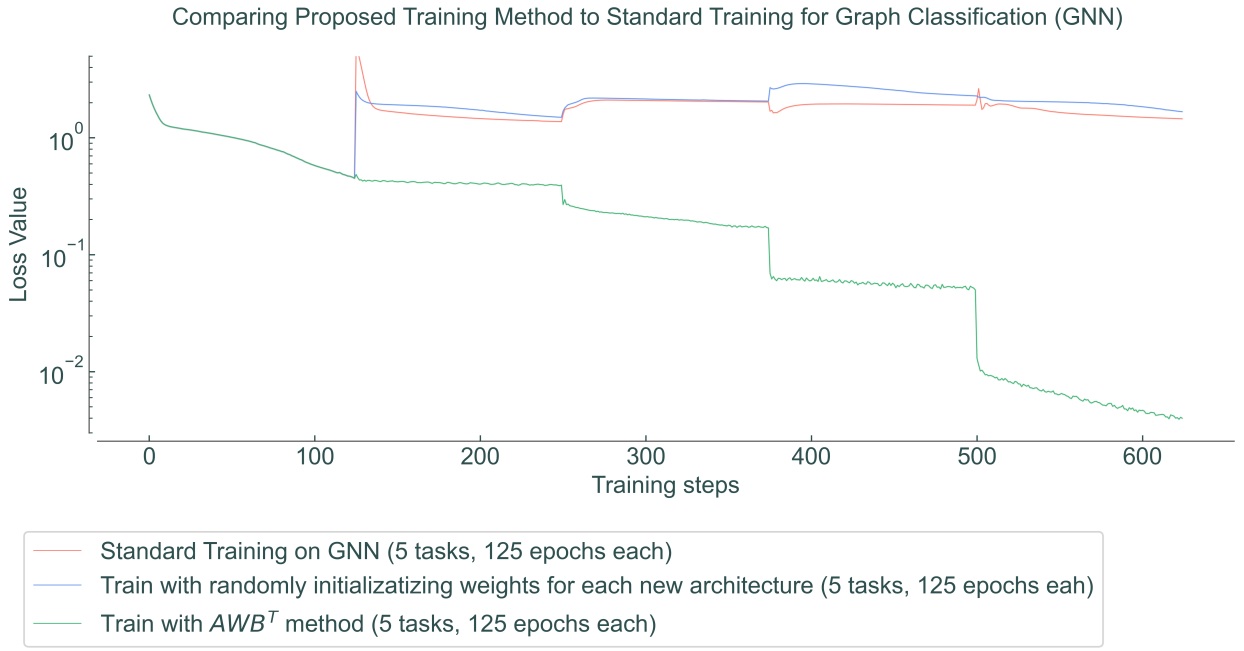


Figure 14: Graph classification experiment: Comparing loss values on training data for baseline continual learning method with method of learning optimal task architecture

Acknowledgments:

The first author AH was supported by the SWARM project, supported by the Department of Energy Award DE-SC0024387. The second author, KR, was supported by the U.S. Department of Energy, Office of Science (SC), Advanced Scientific Computing Research (ASCR), Competitive Portfolios Project on Energy Efficient Computing: A Holistic Methodology, under Contract DE-AC02-06CH11357. We also acknowledge the support by the U.S. Department of Energy for the SciDAC 6 RAPIDS institute. This research used resources of the Argonne Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC02- 06CH11357. We also thank the initial conversation with Dr. Stefan Wild, =Lawrence Berkeley National Laboratory, which led to all of this work.

References

Kunle Adegoke and Olawanle Layeni. The higher derivatives of the inverse tangent function and rapidly convergent BBP-type formulas. URL http://arxiv.org/abs/1603.08540.

Prasanna Balaprakash, Misha Salim, Taylor Uram, Venkatram Vishwanath, and Stefan Wild. DeepHyper: Asynchronous hyperparameter search for deep neural networks. In *2018 IEEE 25th International Conference on High Performance Computing (HiPC)*, pp. 42–51, Bengaluru, India, 2018. IEEE. doi: 10.1109/HiPC.2018.00013.

Dimitri Bertsekas. *Dynamic programming and optimal control: Volume I*, volume 4. Athena scientific, 2012.

Dan Biderman, Jacob Portes, Jose Javier Gonzalez Ortiz, Mansheej Paul, Philip Greengard, Connor Jennings, Daniel King, Sam Havens, Vitaliy Chiley, Jonathan Frankle, et al. LoRAlearns less and forgets less. *arXiv preprint arXiv:2405.09673*, 2024.

Supriyo Chakraborty and Krishnan Raghavan. On understanding of the dynamics of model capacity in continual learning. *arXiv preprint arXiv:2508.08052*, 2025.

Lawrence C Evans. *Partial differential equations*, volume 19. American Mathematical Society, 2022.

Matthias Fey, Jan Eric Lenssen, and contributors. Pytorch geomteric library. PyTorch Geometric, 2025. URL https://pytorch-geometric.readthedocs.io/.

jacob frostig, jascha sohl dickstein, sharad stephens, akintunde adigun, yasaman bahri, noam bard, ben holliday, arnaud doucet, matthew levenberg, alex mayne, aaron van den oord, david pfau, karen simonyan, paul slancman, andy sussex, ashish vadgama, vincent vanhoucke, rory wehnert, and barret zoph. JAX: Composable transformations of Python + NumPy programs, 2018. URL github.com.

Matteo Gambella, Vicente Javier Castro Solar, and Manuel Roveri. SEAL: Searching expandable architectures for incremental learning. *arXiv preprint arXiv:2505.10457*, 2025.

Qiang Gao, Zhipeng Luo, Diego Klabjan, and Fengli Zhang. Efficient architecture search for continual learning. *IEEE Transactions on Neural Networks and Learning Systems*, 34(11):8555–8565, 2022.

Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics (AISTATS)*, pp. 249–256. JMLR Workshop and Conference Proceedings, 2010. URL proceedings.mlr.press.

Zeyu Han, Chao Gao, Jinyang Liu, Jeff Zhang, and Sai Qian Zhang. Parameter-efficient fine-tuning for large models: A comprehensive survey. *arXiv preprint arXiv:2403.14608*, 2024.

Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, Weizhu Chen, et al. Lora: Low-rank adaptation of large language models. *ICLR*, 1(2):3, 2022.

Lei Huang, Jie Qin, Yi Zhou, Fan Zhu, Li Liu, and Ling Shao. Normalization techniques in training dnns: Methodology, analysis and application. *IEEE transactions on Pattern Analysis and Machine Intelligence*, 45(8):10173–10196, 2023.

Patrick Kidger and Cristian Garcia. Equinox: neural networks in JAX via callable PyTrees and filtered transformations. *Differentiable Programming workshop at Neural Information Processing Systems 2021*, 2021.

Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. URL http://arxiv.org/abs/1412.6980.

Tamara G Kolda and Brett W Bader. Tensor decompositions and applications. *SIAM Review*, 51(3):455–500, 2009.

Song Lai, Haohan Zhao, Rong Feng, Changyi Ma, Wenzhuo Liu, Hongbo Zhao, Xi Lin, Dong Yi, Min Xie, Qingfu Zhang, et al. Reinforcement fine-tuning naturally mitigates forgetting in continual post-training. *arXiv preprint arXiv:2507.05386*, 2025.

Brenden M Lake, Ruslan Salakhutdinov, and Joshua B Tenenbaum. Human-level concept learning through probabilistic program induction. *Science*, 350(6266):1332–1338, 2015. doi: 10.1126/science.aab3050.

Jeffrey Larson, Matt Menickelly, and Stefan M Wild. Derivative-free optimization methods. *Acta Numerica*, 28:287–404, 2019.

Jessy Lin, Luke Zettlemoyer, Gargi Ghosh, Wen-Tau Yih, Aram Markosyan, Vincent-Pierre Berges, and Barlas Oğuz. Continual learning via sparse memory finetuning. *arXiv preprint arXiv:2510.15103*, 2025.

Sen Lin, Peizhong Ju, Yingbin Liang, and Ness Shroff. Theory on forgetting and generalization of continual learning. In *International Conference on Machine Learning*, pp. 21078–21100. PMLR, 2023.

Yuqiao Liu, Yanan Sun, Bing Xue, Mengjie Zhang, Gary G Yen, and Kay Chen Tan. A survey on evolutionary neural architecture search. *IEEE Transactions on Neural Networks and Learning Systems*, 34(2):550–570, 2021.

Ilya Loshchilov, Frank Hutter, et al. Fixing weight decay regularization in Adam. *arXiv preprint arXiv:1711.05101*, 5(5):5, 2017.

Aojun Lu, Hangjie Yuan, Tao Feng, and Yanan Sun. Rethinking the stability-plasticity trade-off in continual learning from an architectural perspective. *arXiv preprint arXiv:2506.03951*, 2025.

Yun Luo, Zhen Yang, Fandong Meng, Yafu Li, Jie Zhou, and Yue Zhang. An empirical study of catastrophic forgetting in large language models during continual fine-tuning. *IEEE Transactions on Audio, Speech and Language Processing*, 2025.

Scott Mahan, Emily J. King, and Alex Cloninger. Nonclosedness of sets of neural networks in Sobolev spaces. 137:85–96. ISSN 0893-6080. doi: 10.1016/j.neunet.2021.01.007. URL `https://www.sciencedirect.com/science/article/pii/S0893608021000150`.

Michael McCloskey and Neal J Cohen. Catastrophic interference in connectionist networks: The sequential learning problem. In *Psychology of learning and motivation*, volume 24, pp. 109–165. Elsevier, 1989.

Shishir Muralidhara, Didier Stricker, and René Schuster. CLoRA: Parameter-efficient continual learning with low-rank adaptation. *arXiv preprint arXiv:2507.19887*, 2025.

Ramakanth Pasunuru and Mohit Bansal. Continual and multi-task architecture search. *arXiv preprint arXiv:1906.05226*, 2019.

Philipp Petersen, Mones Raslan, and Felix Voigtlaender. Topological properties of the set of functions generated by neural networks of fixed size. 21(2):375–444. ISSN 1615-3383. doi: 10.1007/s10208-020-09461-0. URL `https://doi.org/10.1007/s10208-020-09461-0`.

Krishnan Raghavan and Prasanna Balaprakash. Formalizing the generalization-forgetting trade-off in continual learning. *Advances in Neural Information Processing Systems*, 34:17284–17297, 2021.

Krishnan Raghavan and Prasanna Balaprakash. Learning continually on a sequence of graphs – the dynamical system way, 2023. URL `https://arxiv.org/abs/2305.12030`.

Walter Rudin. *Principles of mathematical analysis*. McGraw-Hill, 3rd ed. edition, 1976.

Andrei A Rusu, Neil C Rabinowitz, Guillaume Desjardins, Hubert Soyer, James Kirkpatrick, Koray Kavukcuoglu, Razvan Pascanu, and Raia Hadsell. Progressive neural networks. *arXiv preprint arXiv:1606.04671*, 2016.

Hwijae Son, Jin Woo Jang, Woo Jin Han, and Hyung Ju Hwang. Sobolev training for physics informed neural networks. *arXiv preprint arXiv:2101.08932*, 2021.

Nik Weaver. *Measure theory and functional analysis*. World Scientific Publishing Company, 2013.

Martin Wistuba, Prabhu Teja Sivaprasad, Lukas Balles, and Giovanni Zapper. Continual learning with low rank adaptation. *arXiv preprint arXiv:2311.17601*, 2023.

Jaehong Yoon, Eunho Yang, Jeongtae Lee, and Sung Ju Hwang. Lifelong learning with dynamically expandable networks. *arXiv preprint arXiv:1708.01547*, 2017.

A Appendix

B Proofs

B.1 Proof of Lemma 11

Proof 10 *By Lemma 10 and the assumption that* $\mu\left(\bigcup_{\tau=0}^{t}\boldsymbol{\mathcal{X}}(\tau)\Delta\bigcup_{\tau=0}^{t+\Delta t}\boldsymbol{\mathcal{X}}(\tau)\right)\geq\delta$*, we have*

$$\begin{aligned}
E\left(\bigcup_{\tau=0}^{t}\boldsymbol{\mathcal{X}}(\tau)\right) &= E\left(\bigcup_{\tau=0}^{t+\Delta t}\boldsymbol{\mathcal{X}}(\tau)\right)-\Delta t\Big[\mu\left(\bigcup_{\tau=0}^{t}\boldsymbol{\mathcal{X}}(\tau)\Delta\bigcup_{\tau=0}^{t+\Delta t}\boldsymbol{\mathcal{X}}(\tau)\right)\cdot\int_{\bigcup_{\tau=0}^{t}\boldsymbol{\mathcal{X}}(\tau)}\ell(\hat{f}(\mathbf{w},\psi))d\mu\\
&\quad+\int_{\bigcup_{\tau=0}^{t}\boldsymbol{\mathcal{X}}(\tau)}\ell'(\hat{f}(\mathbf{w},\psi))\cdot\partial^1_{\mathbf{w}}\hat{f}(\mathbf{w},\psi)\cdot\Delta w\,d\mu\Big]-o(\Delta t)\\
&= E\left(\bigcup_{\tau=0}^{t+\Delta t}\boldsymbol{\mathcal{X}}(\tau)\right)+\Delta t\left(-\mu\left(\bigcup_{\tau=0}^{t}\boldsymbol{\mathcal{X}}(\tau)\Delta\bigcup_{\tau=0}^{t+\Delta t}\boldsymbol{\mathcal{X}}(\tau)\right)\right)\cdot\int_{\bigcup_{\tau=0}^{t}\boldsymbol{\mathcal{X}}(\tau)\Delta\bigcup_{\tau=0}^{t+\Delta t}\boldsymbol{\mathcal{X}}(\tau)}\ell(\hat{f}(\mathbf{w},\psi))d\mu\\
&\quad-\Delta t\int_{\bigcup_{\tau=0}^{t}\boldsymbol{\mathcal{X}}(\tau)}\ell'(\hat{f}(\mathbf{w},\psi))\cdot\partial^1_{\mathbf{w}}\hat{f}(\mathbf{w},\psi)\cdot\Delta w\,d\mu-o(\Delta t)\\
&\leq E\left(\bigcup_{\tau=0}^{t+\Delta t}\boldsymbol{\mathcal{X}}(\tau)\right)+\Delta t\cdot\delta\cdot\int_{\bigcup_{\tau=0}^{t}\boldsymbol{\mathcal{X}}(\tau)\Delta\bigcup_{\tau=0}^{t+\Delta t}\boldsymbol{\mathcal{X}}(\tau)}\ell(\hat{f}(\mathbf{w},\psi))d\mu\\
&\quad-\Delta t\int_{\bigcup_{\tau=0}^{t}\boldsymbol{\mathcal{X}}(\tau)}\ell'(\hat{f}(\mathbf{w},\psi))\cdot\partial^1_{\mathbf{w}}\hat{f}(\mathbf{w},\psi)\cdot\Delta w\,d\mu-o(\Delta t)
\end{aligned}$$

Subtracting $E\left(\bigcup_{\tau=0}^{t+\Delta t}\boldsymbol{\mathcal{X}}(\tau)\right)$ *from both sides and combining like terms,*

$$\begin{aligned}
E\left(\bigcup_{\tau=0}^{t}\boldsymbol{\mathcal{X}}(\tau)\right)-E\left(\bigcup_{\tau=0}^{t+\Delta t}\boldsymbol{\mathcal{X}}(\tau)\right) &\leq \Delta t\cdot\delta\cdot\int_{\bigcup_{\tau=0}^{t}\boldsymbol{\mathcal{X}}(\tau)\Delta\bigcup_{\tau=0}^{t+\Delta t}\boldsymbol{\mathcal{X}}(\tau)}\ell(\hat{f}(\mathbf{w},\psi))d\mu\\
&\quad-\Delta t\int_{\bigcup_{\tau=0}^{t}\boldsymbol{\mathcal{X}}(\tau)}\ell'(\hat{f}(\mathbf{w},\psi))\cdot\partial^1_{\mathbf{w}}\hat{f}(\mathbf{w},\psi)\cdot\Delta w\,d\mu-o(\Delta t).
\end{aligned}$$

Additionally, we can assume that the loss function ℓ *is bounded above by a constant* M_0*, so*

$$E\left(\bigcup_{\tau=0}^{t}\boldsymbol{\mathcal{X}}(\tau)\right)-E\left(\bigcup_{\tau=0}^{t+\Delta t}\boldsymbol{\mathcal{X}}(\tau)\right)\leq\Delta t\cdot\delta\cdot M_0-\Delta t\int_{\bigcup_{\tau=0}^{t}\boldsymbol{\mathcal{X}}(\tau)}\ell'(\hat{f}(\mathbf{w},\psi))\cdot\partial^1_{\mathbf{w}}\hat{f}(\mathbf{w},\psi)\cdot\Delta w\,d\mu-o(\Delta t).$$

To consider this difference in expected values across future tasks, set $\Delta t=1$ *and let us sum across such future tasks,*

$$\sum_{\tau=t}^{T}\left(E\left(\bigcup_{\nu=0}^{\tau}\boldsymbol{\mathcal{X}}(\nu)\right)-E\left(\bigcup_{\nu=0}^{\tau+1}\boldsymbol{\mathcal{X}}(\nu)\right)\right)\leq\sum_{\tau=t}^{T}\Big(M_0\cdot\delta-\int_{\bigcup_{\nu=0}^{\tau}\boldsymbol{\mathcal{X}}(\nu)}\ell'(\hat{f}(\mathbf{w},\psi))\cdot\partial^1_{\mathbf{w}}\hat{f}(\mathbf{w},\psi)\cdot\Delta w\,d\mu\Big).$$

Now, we can choose the following for each τ

$$\Delta \mathbf{w} = \ell'(\hat{f}(\mathbf{w}, \psi)) \cdot \partial^1_{\mathbf{w}} \hat{f}(\mathbf{w}, \psi).$$

Thus,

$$\sum_{\tau=t}^{T} \left(E\left(\bigcup_{\nu=0}^{\tau} \boldsymbol{\mathcal{X}}(\nu) \right) - E\left(\bigcup_{\nu=0}^{\tau+1} \boldsymbol{\mathcal{X}}(\nu) \right) \right) \leq \sum_{\tau=t}^{T} \Big(M_0 \cdot \delta - \int_{\bigcup_{\nu=0}^{\tau} \boldsymbol{\mathcal{X}}(\nu)} \left(\ell'(\hat{f}(\mathbf{w}, \psi)) \cdot \partial^1_{\mathbf{w}} \hat{f}(\mathbf{w}, \psi) \right)^2 \, d\mu \Big).$$

Now, notice the following

$$J(\mathbf{w}(\tau), \psi(\tau), \boldsymbol{\mathcal{X}}(\tau)) = \int_{\bigcup_{\nu=0}^{\tau} \boldsymbol{\mathcal{X}}(\nu)} \ell(\hat{f}(\mathbf{w}, \psi)) d\mu = E\left(\bigcup_{\nu=0}^{\tau} \boldsymbol{\mathcal{X}}(\nu) \right).$$

Thus,

$$\begin{aligned} \sum_{\tau=t}^{T} (J(\boldsymbol{\mathcal{X}}(\tau)) - J(\boldsymbol{\mathcal{X}}(\tau+1))) &= \sum_{\tau=t}^{T} \left(E\left(\bigcup_{\nu=0}^{\tau} \boldsymbol{\mathcal{X}}(\nu) \right) - E\left(\bigcup_{\nu=0}^{\tau+1} \boldsymbol{\mathcal{X}}(\nu) \right) \right) \\ &\leq \sum_{\tau=t}^{T} \Big(M_0 \cdot \delta - \int_{\bigcup_{\nu=0}^{\tau} \boldsymbol{\mathcal{X}}(\nu)} \left(\ell'(\hat{f}(\mathbf{w}, \psi)) \cdot \partial^1_{\mathbf{w}} \hat{f}(\mathbf{w}, \psi) \right)^2 \, d\mu \Big), \end{aligned}$$

as desired.

B.2 Proof of Lemma 12

Proof 11 *By Lemma 10 and the assumption that $\mu\left(\bigcup_{\tau=0}^{t} \boldsymbol{\mathcal{X}}(\tau) \Delta \bigcup_{\tau=0}^{t+\Delta t} \boldsymbol{\mathcal{X}}(\tau) \right) \geq \delta$, we have*

$$\begin{aligned} E\left(\bigcup_{\tau=0}^{t} \boldsymbol{\mathcal{X}}(\tau) \right) &= E\left(\bigcup_{\tau=0}^{t+\Delta t} \boldsymbol{\mathcal{X}}(\tau) \right) - \Delta t \Big[\mu\left(\bigcup_{\tau=0}^{t} \boldsymbol{\mathcal{X}}(\tau) \Delta \bigcup_{\tau=0}^{t+\Delta t} \boldsymbol{\mathcal{X}}(\tau) \right) \cdot \int_{\bigcup_{\tau=0}^{t} \boldsymbol{\mathcal{X}}(\tau)} \ell(\hat{f}(\mathbf{w}, \psi)) d\mu \\ &\quad + \int_{\bigcup_{\tau=0}^{t} \boldsymbol{\mathcal{X}}(\tau)} \ell'(\hat{f}(\mathbf{w}, \psi)) \cdot \partial^1_{\mathbf{w}} \hat{f}(\mathbf{w}, \psi) \cdot \Delta w \, d\mu \\ &\quad + \int_{\bigcup_{\tau=0}^{t} \boldsymbol{\mathcal{X}}(\tau)} \ell'(\hat{f}(\mathbf{w}, \psi)) \cdot \partial^1_{\psi} \hat{f}(\mathbf{w}, \psi) \cdot \Delta \psi \, d\mu \Big] - o(\Delta t) \\ &= E\left(\bigcup_{\tau=0}^{t+\Delta t} \boldsymbol{\mathcal{X}}(\tau) \right) + \Delta t \left(-\mu\left(\bigcup_{\tau=0}^{t} \boldsymbol{\mathcal{X}}(\tau) \Delta \bigcup_{\tau=0}^{t+\Delta t} \boldsymbol{\mathcal{X}}(\tau) \right) \right) \cdot \int_{\bigcup_{\tau=0}^{t} \boldsymbol{\mathcal{X}}(\tau) \Delta \bigcup_{\tau=0}^{t+\Delta t} \boldsymbol{\mathcal{X}}(\tau)} \ell(\hat{f}(\mathbf{w}, \psi)) d\mu \\ &\quad - \Delta t \int_{\bigcup_{\tau=0}^{t} \boldsymbol{\mathcal{X}}(\tau)} \ell'(\hat{f}(\mathbf{w}, \psi)) \cdot \partial^1_{\mathbf{w}} \hat{f}(\mathbf{w}, \psi) \cdot \Delta w \, d\mu \\ &\quad - \Delta t \int_{\bigcup_{\tau=0}^{t} \boldsymbol{\mathcal{X}}(\tau)} \ell'(\hat{f}(\mathbf{w}, \psi)) \cdot \partial^1_{\psi} \hat{f}(\mathbf{w}, \psi) \cdot \Delta \psi \, d\mu - o(\Delta t) \\ &\leq E\left(\bigcup_{\tau=0}^{t+\Delta t} \boldsymbol{\mathcal{X}}(\tau) \right) + \Delta t \cdot \delta \cdot \int_{\bigcup_{\tau=0}^{t} \boldsymbol{\mathcal{X}}(\tau) \Delta \bigcup_{\tau=0}^{t+\Delta t} \boldsymbol{\mathcal{X}}(\tau)} \ell(\hat{f}(\mathbf{w}, \psi)) d\mu \\ &\quad - \Delta t \int_{\bigcup_{\tau=0}^{t} \boldsymbol{\mathcal{X}}(\tau)} \ell'(\hat{f}(\mathbf{w}, \psi)) \cdot \partial^1_{\mathbf{w}} \hat{f}(\mathbf{w}, \psi) \cdot \Delta w \, d\mu \\ &\quad - \Delta t \int_{\bigcup_{\tau=0}^{t} \boldsymbol{\mathcal{X}}(\tau)} \ell'(\hat{f}(\mathbf{w}, \psi)) \cdot \partial^1_{\psi} \hat{f}(\mathbf{w}, \psi) \cdot \Delta \psi \, d\mu - o(\Delta t). \end{aligned}$$

Subtracting $E\left(\bigcup_{\tau=0}^{t+\Delta t} \boldsymbol{\mathcal{X}}(\tau)\right)$ *from both sides and combining like terms,*

$$\begin{aligned} E\left(\bigcup_{\tau=0}^{t} \boldsymbol{\mathcal{X}}(\tau)\right) - E\left(\bigcup_{\tau=0}^{t+\Delta t} \boldsymbol{\mathcal{X}}(\tau)\right) \leq \Delta t \cdot \delta \cdot \int_{\bigcup_{\tau=0}^{t} \boldsymbol{\mathcal{X}}(\tau) \Delta \bigcup_{\tau=0}^{t+\Delta t} \boldsymbol{\mathcal{X}}(\tau)} \ell(\hat{f}(\mathbf{w}, \psi)) d\mu \\ - \Delta t \int_{\bigcup_{\tau=0}^{t} \boldsymbol{\mathcal{X}}(\tau)} \ell'(\hat{f}(\mathbf{w}, \psi)) \cdot \partial_{\mathbf{w}}^{1} \hat{f}(\mathbf{w}, \psi) \cdot \Delta w \, d\mu \\ - \Delta t \int_{\bigcup_{\tau=0}^{t} \boldsymbol{\mathcal{X}}(\tau)} \ell'(\hat{f}(\mathbf{w}, \psi)) \cdot \partial_{\psi}^{1} \hat{f}(\mathbf{w}, \psi) \cdot \Delta \psi \, d\mu - o(\Delta t). \end{aligned}$$

Additionally, we can assume that the loss function ℓ *is bounded above by a constant* M_0*, so*

$$\begin{aligned} E\left(\bigcup_{\tau=0}^{t} \boldsymbol{\mathcal{X}}(\tau)\right) - E\left(\bigcup_{\tau=0}^{t+\Delta t} \boldsymbol{\mathcal{X}}(\tau)\right) \leq \Delta t \cdot \delta \cdot M_0 - \Delta t \int_{\bigcup_{\tau=0}^{t} \boldsymbol{\mathcal{X}}(\tau)} \ell'(\hat{f}(\mathbf{w}, \psi)) \cdot \partial_{\mathbf{w}}^{1} \hat{f}(\mathbf{w}, \psi) \cdot \Delta w \, d\mu \\ - \Delta t \int_{\bigcup_{\tau=0}^{t} \boldsymbol{\mathcal{X}}(\tau)} \ell'(\hat{f}(\mathbf{w}, \psi)) \cdot \partial_{\psi}^{1} \hat{f}(\mathbf{w}, \psi) \cdot \Delta \psi \, d\mu - o(\Delta t). \end{aligned}$$

To consider this difference in expected values across future tasks, set $\Delta t = 1$ *and let us sum across such future tasks,*

$$\begin{aligned} \sum_{\tau=t}^{T} \left(E\left(\bigcup_{\nu=0}^{\tau} \boldsymbol{\mathcal{X}}(\nu)\right) - E\left(\bigcup_{\nu=0}^{\tau+1} \boldsymbol{\mathcal{X}}(\nu)\right)\right) \leq \sum_{\tau=t}^{T} \Big(M_0 \cdot \delta - \int_{\bigcup_{\nu=0}^{\tau} \boldsymbol{\mathcal{X}}(\nu)} \ell'(\hat{f}(\mathbf{w}, \psi)) \cdot \partial_{\mathbf{w}}^{1} \hat{f}(\mathbf{w}, \psi) \cdot \Delta w \, d\mu \\ - \int_{\bigcup_{\tau=0}^{t} \boldsymbol{\mathcal{X}}(\tau)} \ell'(\hat{f}(\mathbf{w}, \psi)) \cdot \partial_{\psi}^{1} \hat{f}(\mathbf{w}, \psi) \cdot \Delta \psi \, d\mu \Big). \end{aligned}$$

Now, we can choose the following for each ν

$$\Delta \mathbf{w} = \ell'(\hat{f}(\mathbf{w}, \psi)) \cdot \partial_{\mathbf{w}}^{1} \hat{f}(\mathbf{w}, \psi).$$

and

$$\Delta \psi = \ell'(\hat{f}(\mathbf{w}, \psi)) \cdot \partial_{\psi}^{1} \hat{f}(\mathbf{w}, \psi).$$

Thus,

$$\begin{aligned} \sum_{\tau=t}^{T} \left(E\left(\bigcup_{\nu=0}^{\tau} \boldsymbol{\mathcal{X}}(\nu)\right) - E\left(\bigcup_{\nu=0}^{\tau+1} \boldsymbol{\mathcal{X}}(\nu)\right)\right) \leq \sum_{\tau=t}^{T} \Big(M_0 \cdot \delta - \int_{\bigcup_{\nu=0}^{\tau} \boldsymbol{\mathcal{X}}(\nu)} \left(\ell'(\hat{f}(\mathbf{w}, \psi)) \cdot \partial_{\mathbf{w}}^{1} \hat{f}(\mathbf{w}, \psi) \right)^2 d\mu \\ - \int_{\bigcup_{\nu=0}^{\tau} \boldsymbol{\mathcal{X}}(\nu)} \left(\ell'(\hat{f}(\mathbf{w}, \psi)) \cdot \partial_{\psi}^{1} \hat{f}(\mathbf{w}, \psi) \right)^2 d\mu \Big). \end{aligned}$$

Now, notice the following

$$J(\mathbf{w}(\tau), \psi(\tau), \boldsymbol{\mathcal{X}}(\tau)) = \int_{\bigcup_{\nu=0}^{\tau} \boldsymbol{\mathcal{X}}(\nu)} \ell(\hat{f}(\mathbf{w}, \psi)) d\mu = E\left(\bigcup_{\nu=0}^{\tau} \boldsymbol{\mathcal{X}}(\nu)\right).$$

Thus,

$$\begin{aligned} \sum_{\tau=t}^{T} \left(J(\boldsymbol{\mathcal{X}}(\tau)) - J(\boldsymbol{\mathcal{X}}(\tau+1)) \right) &= \sum_{\tau=t}^{T} \left(E\left(\bigcup_{\nu=0}^{\tau} \boldsymbol{\mathcal{X}}(\nu)\right) - E\left(\bigcup_{\nu=0}^{\tau+1} \boldsymbol{\mathcal{X}}(\nu)\right)\right) \\ &\leq \sum_{\tau=t}^{T} \Big(M_0 \cdot \delta - \int_{\bigcup_{\nu=0}^{\tau} \boldsymbol{\mathcal{X}}(\nu)} \left(\ell'(\hat{f}(\mathbf{w}, \psi)) \cdot \partial_{\mathbf{w}}^{1} \hat{f}(\mathbf{w}, \psi) \right)^2 d\mu \\ &\quad - \int_{\bigcup_{\nu=0}^{\tau} \boldsymbol{\mathcal{X}}(\nu)} \left(\ell'(\hat{f}(\mathbf{w}, \psi)) \cdot \partial_{\psi}^{1} \hat{f}(\mathbf{w}, \psi) \right)^2 d\mu \Big) \end{aligned}$$

as desired.

B.3 Proof of Proposition 14

Proof 12 *Let J, V, and V^* be as defined above. To begin, we split the sum in $V(t)$ over the discrete intervals $[t, t+\Delta t]$ and $[t+\Delta t, T]$. Observe,*

$$\begin{aligned}
V^*(t) &= \min_{w \in \mathcal{W}(\psi^*(t))} \int_t^T J(w(\tau), \psi^*(t), \boldsymbol{\mathcal{X}}(\tau)) d\tau \\
&= \min_{w \in \mathcal{W}(\psi^*(t))} \left[\int_t^{t+\Delta t} J(w(\tau), \psi^*(t), \boldsymbol{\mathcal{X}}(\tau)) d\tau + \int_{t+\Delta t}^T J(w(\tau), \psi^*(t), \boldsymbol{\mathcal{X}}(\tau)) d\tau \right] \\
&= \min_{w \in \mathcal{W}(\psi^*(t))} \int_t^{t+\Delta t} J(w(\tau), \psi^*(t), \boldsymbol{\mathcal{X}}(\tau)) d\tau + V^*(t+\Delta t) \\
&= \min_{w \in \mathcal{W}(\psi^*(t))} J(\mathbf{w}(t), \psi^*(t), \boldsymbol{\mathcal{X}}(t)) \Delta t + V^*(t+\Delta t). \qquad (16)
\end{aligned}$$

Now, we provide the Taylor series expansion of $V^*(t+\Delta t)$ about t. Notice,

$$\begin{aligned}
V^*(t+\Delta t) &= V^*(t) + \Delta t \left[\partial_t^1 V^*(t) + \partial_{\boldsymbol{\mathcal{X}}}^1 V^*(t) d_t \boldsymbol{\mathcal{X}} + \partial_{\mathbf{w}}^1 V^*(t) \partial_t^1 \mathbf{w} \right] + o(\Delta t) \\
&= V^*(t) + \Delta t \left[\partial_t^1 V^*(t) + \partial_{\boldsymbol{\mathcal{X}}}^1 V^*(t) d_t \boldsymbol{\mathcal{X}} + \partial_{\mathbf{w}}^1 V^*(t) [\mathrm{A}^*(t) \mathbf{w}(t) (\mathrm{B}^*(t))^T + u(t)] \right] + o(\Delta t), \qquad (17)
\end{aligned}$$

where $u(t)$ represents the updates made to the each weights matrix of the new dimensions. Substituting 17 into 16, we have

$$\begin{aligned}
V^*(t) = \min_{w \in \mathcal{W}(\psi^*(t))} & J(\mathbf{w}(t), \psi^*(t), \boldsymbol{\mathcal{X}}(t)) \Delta t + V^*(t) + \Delta t \Big[\partial_t^1 V^*(t) + \partial_{\boldsymbol{\mathcal{X}}}^1 V^*(t) d_t \boldsymbol{\mathcal{X}} \\
& + \partial_{\mathbf{w}}^1 V^*(t) [\mathrm{A}^*(t) \mathbf{w}(t) (\mathrm{B}^*(t))^T + u(t)] \Big] + o(\Delta t).
\end{aligned}$$

Canceling $V^*(t)$ gives

$$\begin{aligned}
0 = \min_{w \in \mathcal{W}(\psi^*(t))} & J(\mathbf{w}(t), \psi^*(t), \boldsymbol{\mathcal{X}}(t)) \Delta t + \Delta t \left[\partial_t^1 V^*(t) + \partial_{\boldsymbol{\mathcal{X}}}^1 V^*(t) d_t \boldsymbol{\mathcal{X}} \right. \\
& \left. + \partial_{\mathbf{w}}^1 V^*(t) [\mathrm{A}^*(t) \mathbf{w}(t) (\mathrm{B}^*(t))^T + u(t)] \right] + o(\Delta t).
\end{aligned}$$

Dividing both sides by Δt produces

$$0 = \min_{w \in \mathcal{W}(\psi^*(t))} J(\mathbf{w}(t), \psi^*(t), \boldsymbol{\mathcal{X}}(t)) + \partial_t^1 V^*(t) + \partial_{\boldsymbol{\mathcal{X}}}^1 V^*(t) V^*(t) d_t \boldsymbol{\mathcal{X}} + \partial_{\mathbf{w}}^1 V^*(t) [\mathrm{A}^*(t) \mathbf{w}(t) (\mathrm{B}^*(t))^T + u(t)].$$

Finally, reordering gives

$$-\partial_t^1 V^*(t) = \min_{w \in \mathcal{W}(\psi^*(t))} J(\mathbf{w}(t), \psi^*(t), \boldsymbol{\mathcal{X}}(t)) + \partial_{\boldsymbol{\mathcal{X}}}^1 V^*(t) d_t \boldsymbol{\mathcal{X}} + \partial_{\mathbf{w}}^1 V^*(t) V^*(t) [\mathrm{A}^*(t) \mathbf{w}(t) (\mathrm{B}^*(t))^T + u(t)],$$

as desired.

B.4 Proof of Theorem 15

Proof 13 *Given proposition 14, assume there exists an stochastic gradient based optimization procedure and choose $u(t) = -\sum^I \alpha(i) \mathrm{g}^{(i)}$ such that $\min_{\mathbf{w} \in \mathcal{W}(\psi^*(t))} J(\mathbf{w}(t), \psi^*(t), \boldsymbol{\mathcal{X}}(t)) \leq \varepsilon$, where I is the number of updates. Then, we have*

$$-\partial_t^1 V^*(t) \leq \varepsilon + \partial_{\boldsymbol{\mathcal{X}}}^1 V^*(t) d_t \boldsymbol{\mathcal{X}} + \partial_{\mathbf{w}}^1 V^*(t) \left(\mathrm{A}^*(t) \mathbf{w}(t) (\mathrm{B}^*(t))^T - \sum^I \alpha(i) \mathrm{g}^{(i)} \right)$$

For any dt in terms of tasks t let $\partial^1_{\boldsymbol{\mathcal{X}}} V^*(t) d_t \boldsymbol{\mathcal{X}} \leq \left\|\partial^1_{\boldsymbol{\mathcal{X}}} V^*(t)\right\|_{W^{k,p}(\mathrm{D})} \|d_t \boldsymbol{\mathcal{X}}\|_{W^{k,p}(\mathrm{D})} \leq \rho^{(\mathrm{x})}_{\mathrm{MAX}} \delta$ *as* $\|d_t \boldsymbol{\mathcal{X}}\|_{W^{k,p}(\mathrm{D})} \geq \mu(\boldsymbol{\mathcal{X}}(t) \triangle \boldsymbol{\mathcal{X}}(t+1)) \geq \delta$ *and* $\left\|\partial^1_{\boldsymbol{\mathcal{X}}} V^*(t)\right\|_{W^{k,p}(\mathrm{D})}$ *is less than the largest singular value of* $\partial^1_{\boldsymbol{\mathcal{X}}} V^*(t)$*. Thus we write*

$$-\partial^1_t V^*(t) \leq \varepsilon + \rho^{(\mathrm{x})}_{\mathrm{MAX}} \delta + \partial^1_{\mathbf{w}} V^*(t) \left(\mathrm{A}^*(t) \mathbf{w}(t) (\mathrm{B}^*(t))^T - \sum^{I} \alpha(i) \mathrm{g}^{(i)} \right)$$

Taking g_{MIN} *to be the smallest value of the gradient over all update iterations, we have*

$$\begin{aligned}
-\partial^1_t &\leq \varepsilon + \rho^{(\mathrm{x})}_{\mathrm{MAX}} \delta + \partial^1_{\mathbf{w}} V^*(t) \left(\mathrm{A}^*(t) \mathbf{w}(t) (\mathrm{B}^*(t))^T - \mathrm{g}_{\mathrm{MIN}} \sum^{I} \alpha(i) \right) \\
&\leq \varepsilon + \rho^{(\mathrm{x})}_{\mathrm{MAX}} \delta + \left\|\partial^1_{\mathbf{w}} V^*(t)\right\|_{W^{k,p}(\mathrm{D})} \left\|\mathrm{A}^*(t) \mathbf{w}(t) (\mathrm{B}^*(t))^T\right\|_{W^{k,p}(\mathrm{D})} - \partial^1_{\mathbf{w}} V^*(t) \left(\mathrm{g}_{\mathrm{MIN}} \sum^{I} \alpha(i) \right) \\
&\leq \varepsilon + \left[\rho^{(\mathrm{x})}_{\mathrm{MAX}} \delta + \rho^{(\mathbf{w})}_{\mathrm{MAX}} \left\|\mathrm{A}^*(t) \mathbf{w}(t) (\mathrm{B}^*(t))^T\right\|_{W^{k,p}(\mathrm{D})} - \rho^{(\mathbf{w})}_{\mathrm{MIN}} \|\mathrm{g}_{\mathrm{MIN}}\|_{W^{k,p}(\mathrm{D})} \left\| \sum^{I} \alpha(i) \right\|_{W^{k,p}(\mathrm{D})} \right]
\end{aligned}$$

This finally provides

$$\partial^1_t \leq \sup\varepsilon - \inf \left[\rho^{(\mathrm{x})}_{\mathrm{MAX}} \delta + \rho^{(\mathbf{w})}_{\mathrm{MAX}} \left\|\mathrm{A}^*(t) \mathbf{w}(t) (\mathrm{B}^*(t))^T\right\|_{W^{k,p}(\mathrm{D})} - \rho^{(\mathbf{w})}_{\mathrm{MIN}} \|\mathrm{g}_{\mathrm{MIN}}\|_{W^{k,p}(\mathrm{D})} \left\| \sum^{I} \alpha(i) \right\|_{W^{k,p}(\mathrm{D})} \right]$$

For the total variation to be upper bounded, we need the quantity in the brackets to go to zero, this provides.

$$\left[\rho^{(\mathrm{x})}_{\mathrm{MAX}} \delta + \rho^{(\mathbf{w})}_{\mathrm{MAX}} \left\|\mathrm{A}^*(t) \mathbf{w}(t) (\mathrm{B}^*(t))^T\right\|_{W^{k,p}(\mathrm{D})} - \rho^{(\mathbf{w})}_{\mathrm{MIN}} \|\mathrm{g}_{\mathrm{MIN}}\|_{W^{k,p}(\mathrm{D})} \left\| \sum^{I} \alpha(i) \right\|_{W^{k,p}(\mathrm{D})} \right] = 0$$

$$\rho^{(\mathrm{x})}_{\mathrm{MAX}} \delta = \left[\rho^{(\mathbf{w})}_{\mathrm{MIN}} \|\mathrm{g}_{\mathrm{MIN}}\|_{W^{k,p}(\mathrm{D})} \left\| \sum^{I} \alpha(i) \right\|_{W^{k,p}(\mathrm{D})} - \rho^{(\mathbf{w})}_{\mathrm{MAX}} \left\|\mathrm{A}^*(t) \mathbf{w}(t) (\mathrm{B}^*(t))^T\right\|_{W^{k,p}(\mathrm{D})} \right]$$

providing the result

C Introduction

C.1 AWB Transformation Principle

The Adaptive Weight Basis (AWB) enables architecture morphing during continual learning through the transformation:

$$\mathbf{V} = \mathbf{AWB}^T \tag{18}$$

where:

- $\mathbf{W}$: Original weight matrix from the previous architecture
- $\mathbf{A}$: Output transformation matrix
- $\mathbf{B}$: Input transformation matrix
- $\mathbf{V}$: New weight matrix in the expanded architecture

C.2 Key Invariant

Architecture morphing rule: Input and output dimensions must be preserved, while hidden layer dimensions can be expanded.

C.3 Notation

For a layer with weight matrix $\mathbf{W} \in \mathbb{R}^{m \times n}$:

- n = input dimension (number of input features)
- m = output dimension (number of output features)
- Forward pass: $\mathbf{y} = \mathbf{W}\mathbf{x}$ where $\mathbf{x} \in \mathbb{R}^n$, $\mathbf{y} \in \mathbb{R}^m$

D MLP AWB Calculation

D.1 Architecture Specification

Original architecture:

$$\text{sizes} = [64, 128, 128, 10] \tag{19}$$

This defines a 3-layer network:

- Layer 0: $64 \to 128$ (input layer to first hidden)
- Layer 1: $128 \to 128$ (first hidden to second hidden)
- Layer 2: $128 \to 10$ (second hidden to output)

AWB architecture:

$$\text{new\_arch} = [64, 256, 256, 10] \tag{20}$$

This expands hidden layers while preserving input (64) and output (10):

- Layer 0: $64 \to 256$
- Layer 1: $256 \to 256$
- Layer 2: $256 \to 10$

D.2 Original Weight Matrices

$$\mathbf{W}_0 \in \mathbb{R}^{128 \times 64} \quad (\text{maps } 64 \to 128) \tag{21}$$
$$\mathbf{W}_1 \in \mathbb{R}^{128 \times 128} \quad (\text{maps } 128 \to 128) \tag{22}$$
$$\mathbf{W}_2 \in \mathbb{R}^{10 \times 128} \quad (\text{maps } 128 \to 10) \tag{23}$$

D.3 AWB Transformation Matrices

The transformation matrices are constructed as:

$$\mathbf{A}_i \in \mathbb{R}^{\text{new\_out}_i \times \text{old\_out}_i} \tag{24}$$
$$\mathbf{B}_i \in \mathbb{R}^{\text{new\_in}_i \times \text{old\_in}_i} \tag{25}$$

For each layer:

Layer 0:

$$\mathbf{A}_0 \in \mathbb{R}^{256 \times 128} \quad (\text{output: } 128 \to 256) \tag{26}$$
$$\mathbf{B}_0 \in \mathbb{R}^{64 \times 64} \quad (\text{input: } 64 \to 64, \text{ identity-like}) \tag{27}$$

Layer 1:

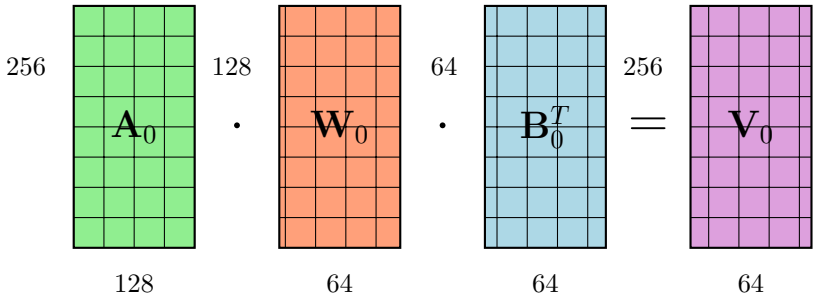
$$\mathbf{A}_1 \in \mathbb{R}^{256 \times 128} \quad (\text{output: } 128 \to 256) \tag{28}$$
$$\mathbf{B}_1 \in \mathbb{R}^{256 \times 128} \quad (\text{input: } 128 \to 256) \tag{29}$$

Layer 2:

$$\mathbf{A}_2 \in \mathbb{R}^{10 \times 10} \quad (\text{output: } 10 \to 10, \text{ identity-like}) \tag{30}$$
$$\mathbf{B}_2 \in \mathbb{R}^{256 \times 128} \quad (\text{input: } 128 \to 256) \tag{31}$$

D.4 Layer 0 Transformation

$$\mathbf{V}_0 = \mathbf{A}_0 \mathbf{W}_0 \mathbf{B}_0^T \tag{32}$$

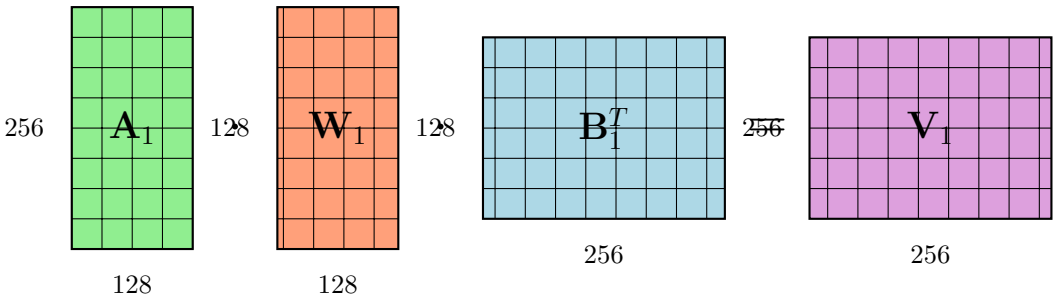


Dimension verification:

$$\begin{aligned} \mathbf{V}_0 &= \mathbf{A}_0 \mathbf{W}_0 \mathbf{B}_0^T && (33)\\ &= (256 \times 128) \cdot (128 \times 64) \cdot (64 \times 64) && (34)\\ &= (256 \times 64) \quad \checkmark && (35) \end{aligned}$$

D.5 Layer 1 Transformation

$$\mathbf{V}_1 = \mathbf{A}_1 \mathbf{W}_1 \mathbf{B}_1^T \tag{36}$$

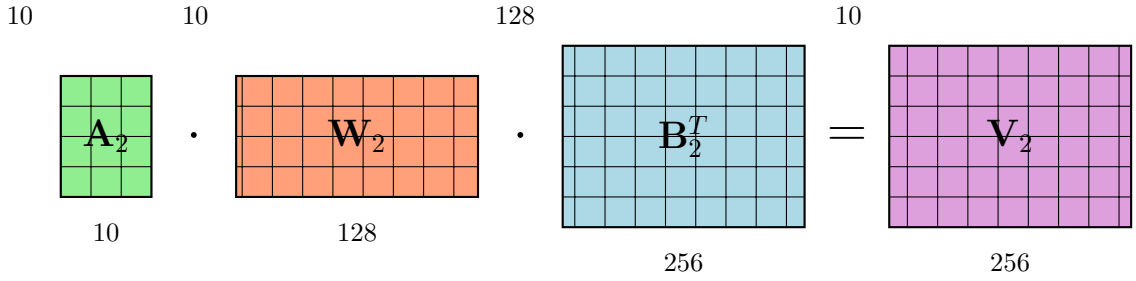


Dimension verification:

$$\begin{aligned} \mathbf{V}_1 &= \mathbf{A}_1 \mathbf{W}_1 \mathbf{B}_1^T && (37)\\ &= (256 \times 128) \cdot (128 \times 128) \cdot (128 \times 256) && (38)\\ &= (256 \times 256) \quad \checkmark && (39) \end{aligned}$$

D.6 Layer 2 Transformation

$$\mathbf{V}_2 = \mathbf{A}_2 \mathbf{W}_2 \mathbf{B}_2^T \tag{40}$$



Dimension verification:

$$\mathbf{V}_2 = \mathbf{A}_2\mathbf{W}_2\mathbf{B}_2^T \tag{41}$$
$$= (10 \times 10) \cdot (10 \times 128) \cdot (128 \times 256) \tag{42}$$
$$= (10 \times 256) \quad \checkmark \tag{43}$$

D.7 Summary

Layer	Original $\mathbf{W}$	AWB $\mathbf{A}$	AWB $\mathbf{B}$	Result $\mathbf{V}$
0	128×64	256×128	64×64	256×64
1	128×128	256×128	256×128	256×256
2	10×128	10×10	256×128	10×256

Table 2: MLP AWB transformation dimensions

Key observations:

- Input dimension preserved: $64 \to 64$ (Layer 0 input)
- Output dimension preserved: $10 \to 10$ (Layer 2 output)
- Hidden layer expanded: $128 \to 256$ (Layers 0, 1, 2 hidden dimensions)

E CNN3D AWB Calculation

CNN3D processes 3-channel images (e.g., CIFAR) with both convolutional and feed-forward layers.

E.1 Feed-Forward Layers

Original architecture:

$$\text{feed\_sizes} = [2304, 256, 10] \tag{44}$$

After two conv+pool layers, the flattened feature size is 2304. This feeds into:

- Layer 0: $2304 \to 256$
- Layer 1: $256 \to 10$

AWB architecture:

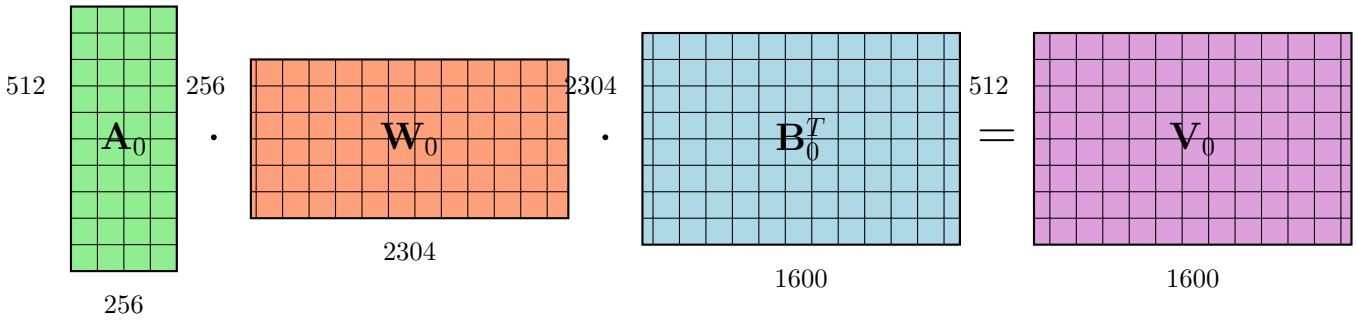
$$\text{new\_arch} = [1600, 512, 10] \tag{45}$$

With expanded filters (5×5 instead of 3×3), the flattened size becomes 1600:

- Layer 0: $1600 \to 512$
- Layer 1: $512 \to 10$

E.2 Feed Layer 0 Transformation

$$\mathbf{V}_0 = \mathbf{A}_0 \mathbf{W}_0 \mathbf{B}_0^T \tag{46}$$

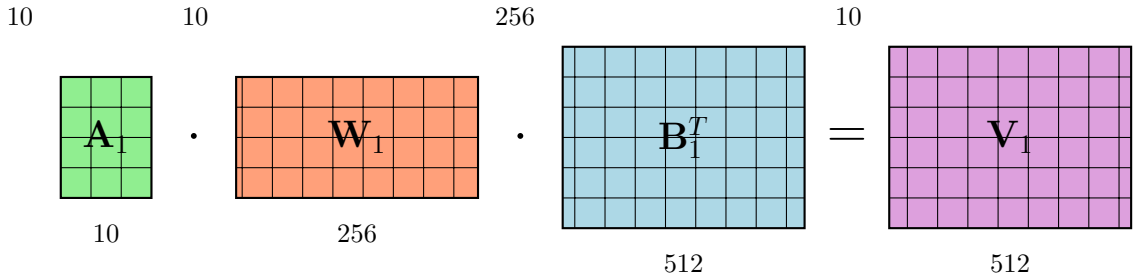


Dimension verification:

$$\begin{aligned} \mathbf{V}_0 &= \mathbf{A}_0 \mathbf{W}_0 \mathbf{B}_0^T && (47)\\ &= (512 \times 256) \cdot (256 \times 2304) \cdot (2304 \times 1600) && (48)\\ &= (512 \times 1600) \quad \checkmark && (49) \end{aligned}$$

E.3 Feed Layer 1 Transformation

$$\mathbf{V}_1 = \mathbf{A}_1 \mathbf{W}_1 \mathbf{B}_1^T \tag{50}$$



Dimension verification:

$$\mathbf{V}_1 = \mathbf{A}_1\mathbf{W}_1\mathbf{B}_1^T \tag{51}$$
$$= (10 \times 10) \cdot (10 \times 256) \cdot (256 \times 512) \tag{52}$$
$$= (10 \times 512) \quad \checkmark \tag{53}$$

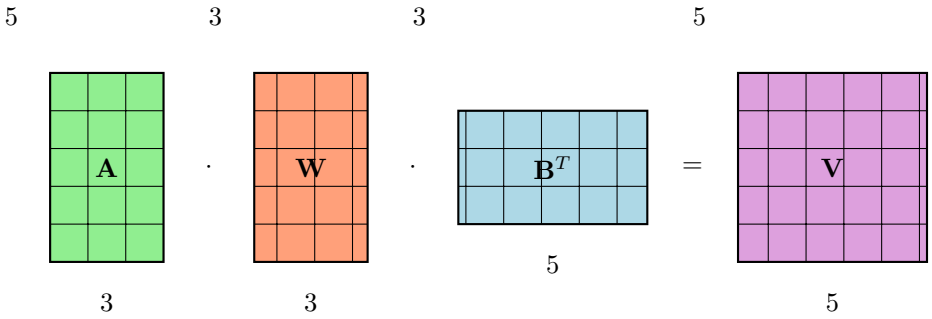
E.4 Convolutional Filter Transformation

For a single convolutional filter $\mathbf{W}_{\text{filter}}^{(i,c)}$ where i is the output channel and c is the input channel:

Original filter: 3×3 kernel

AWB filter: 5×5 kernel

$$\mathbf{V}_{\text{filter}}^{(i,c)} = \mathbf{A}_{\text{conv}}^{(i,c)}\mathbf{W}_{\text{filter}}^{(i,c)}\mathbf{B}_{\text{conv}}^{(i,c)T} \tag{54}$$



Dimension verification:

$$\mathbf{V}_{\text{filter}} = \mathbf{A}\mathbf{W}_{\text{filter}}\mathbf{B}^T \tag{55}$$
$$= (5 \times 3) \cdot (3 \times 3) \cdot (3 \times 5) \tag{56}$$
$$= (5 \times 5) \quad \checkmark \tag{57}$$

This transformation is applied to each filter in each convolutional layer, expanding the spatial kernel size from 3×3 to 5×5.

F CNN AWB Calculation

CNN processes single-channel images (e.g., MNIST) with one convolutional layer followed by feed-forward layers.

F.1 Feed-Forward Layers

Original architecture:

$$\text{feed\_sizes} = [1728, 64, 10] \tag{58}$$

After conv+pool, the flattened size is 1728:

- Layer 0: $1728 \to 64$
- Layer 1: $64 \to 10$

AWB architecture:

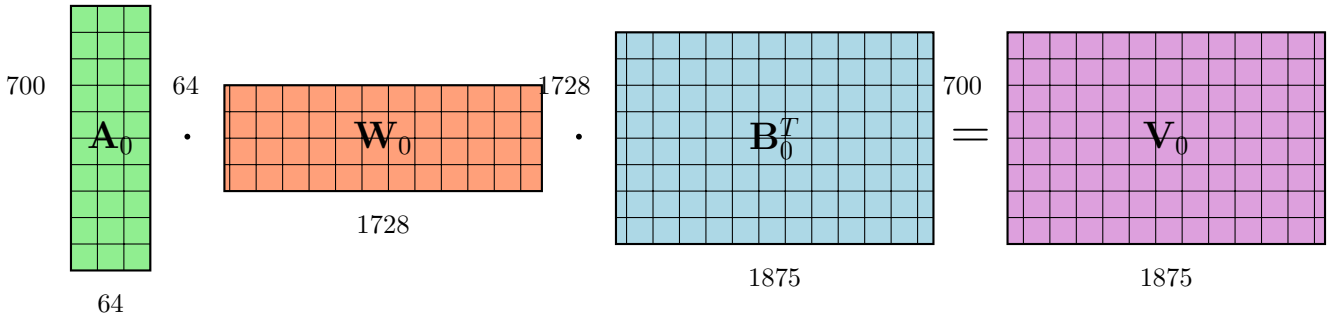
$$\text{new\_arch} = [1875, 700, 10] \tag{59}$$

With AWB filters:

- Layer 0: $1875 \to 700$
- Layer 1: $700 \to 10$

F.2 Feed Layer 0 Transformation

$$\mathbf{V}_0 = \mathbf{A}_0 \mathbf{W}_0 \mathbf{B}_0^T \tag{60}$$

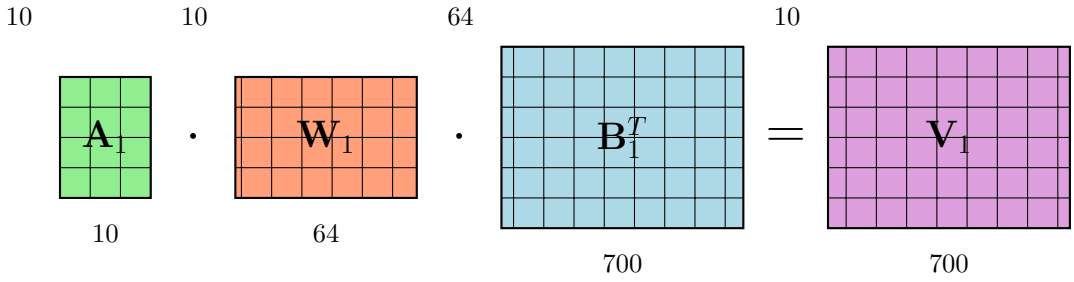


Dimension verification:

$$\begin{aligned} \mathbf{V}_0 &= \mathbf{A}_0 \mathbf{W}_0 \mathbf{B}_0^T && (61)\\ &= (700 \times 64) \cdot (64 \times 1728) \cdot (1728 \times 1875) && (62)\\ &= (700 \times 1875) \quad \checkmark && (63) \end{aligned}$$

F.3 Feed Layer 1 Transformation

$$\mathbf{V}_1 = \mathbf{A}_1 \mathbf{W}_1 \mathbf{B}_1^T \tag{64}$$



Dimension verification:

$$\mathbf{V}_1 = \mathbf{A}_1\mathbf{W}_1\mathbf{B}_1^T \tag{65}$$
$$= (10 \times 10) \cdot (10 \times 64) \cdot (64 \times 700) \tag{66}$$
$$= (10 \times 700) \quad \checkmark \tag{67}$$

G GCN AWB Calculation

Graph Convolutional Networks (GCN) consist of graph convolutional layers followed by feed-forward layers for graph classification.

G.1 Architecture Specification

GCN layers:

$$\text{gcn\_sizes} = [5, 64] \tag{68}$$

Single GCN layer: $5 \to 64$ (5 input node features, 64 output features)

Feed-forward layers:

$$\text{feed\_sizes} = [64, 32, 16, 10] \tag{69}$$

Three feed layers after graph pooling:

- Layer 0: $64 \to 32$
- Layer 1: $32 \to 16$
- Layer 2: $16 \to 10$

AWB architectures:

$$\text{awb\_gcn\_arch} = [5, 64] \quad \text{(preserved)} \tag{70}$$

$$\text{awb\_fnn\_arch} = [64, 140, 140, 10] \tag{71}$$

Note: GCN layer dimensions are preserved (no expansion), while feed layers expand hidden dimensions.

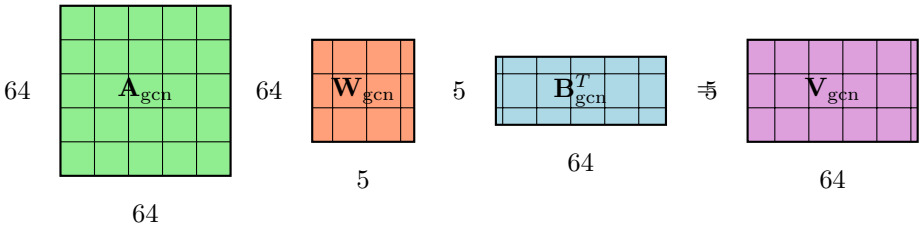
G.2 GCN Layer Transformation

GCN forward pass: $\mathbf{y} = \mathbf{A}_{\text{adj}}(\mathbf{xW})$

where $\mathbf{A}_{\text{adj}}$ is the normalized adjacency matrix.

Since AWB preserves dimensions: $5 \to 64$ remains $5 \to 64$

$$\mathbf{V}_{\text{gcn}} = \mathbf{A}_{\text{gcn}}\mathbf{W}_{\text{gcn}}\mathbf{B}_{\text{gcn}}^T \tag{72}$$



Dimension verification:

$$\mathbf{V}_{\text{gcn}} = \mathbf{A}_{\text{gcn}}\mathbf{W}_{\text{gcn}}\mathbf{B}_{\text{gcn}}^T \tag{73}$$

$$= (64 \times 64) \cdot (64 \times 5) \cdot (5 \times 64) \tag{74}$$

$$= (64 \times 64) \text{ intermediate} \cdot (64 \times 5) \tag{75}$$

$$= (5 \times 64) \quad \checkmark \quad \text{(dimensions preserved)} \tag{76}$$

Wait, let me recalculate. For GCN, weight matrix has shape (in_size, out_size) = (5, 64).

With $\mathbf{A} \in \mathbb{R}^{64\times64}$ and $\mathbf{B} \in \mathbb{R}^{64\times5}$:

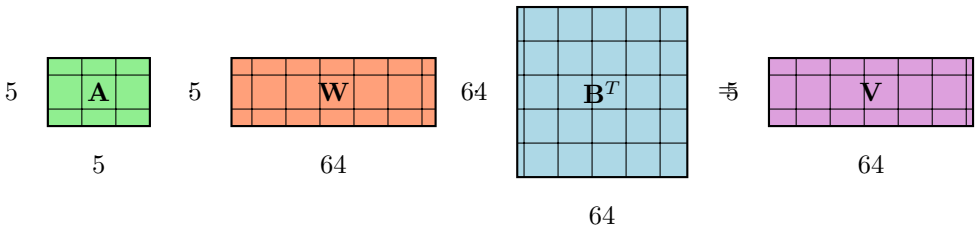
$$\mathbf{V}_{\text{gcn}} = \mathbf{A}_{\text{gcn}}\mathbf{W}_{\text{gcn}}\mathbf{B}_{\text{gcn}}^T \tag{77}$$

$$= (64 \times 64) \cdot (5 \times 64) \quad \text{dimension mismatch!} \tag{78}$$

Correction: For GCN layer preserving (5×64), we need:

$$\mathbf{A}_{\text{gcn}} \in \mathbb{R}^{5\times5} \tag{79}$$

$$\mathbf{B}_{\text{gcn}} \in \mathbb{R}^{64\times64} \tag{80}$$



Corrected dimension verification:

$$\mathbf{V}_{\text{gcn}} = \mathbf{A}_{\text{gcn}}\mathbf{W}_{\text{gcn}}\mathbf{B}_{\text{gcn}}^T \tag{81}$$

$$= (5 \times 5) \cdot (5 \times 64) \cdot (64 \times 64) \tag{82}$$

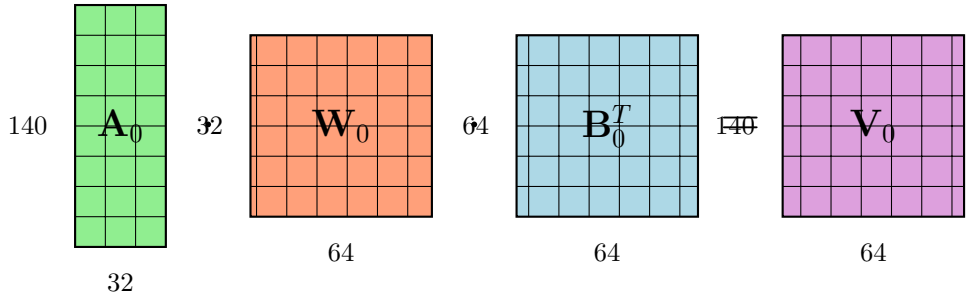
$$= (5 \times 64) \quad \checkmark \quad \text{(dimensions preserved)} \tag{83}$$

G.3 Feed Layer 0 Transformation

Note: Feed layers use Linear3 with weight shape (out_size, in_size), so forward pass is $\mathbf{y} = \mathbf{x}\mathbf{W}^T + \mathbf{b}$.

$$\mathbf{V}_0 = \mathbf{A}_0\mathbf{W}_0\mathbf{B}_0^T \tag{84}$$

Original: (32×64), AWB: (140×64)



Dimension verification:

$$\mathbf{V}_0 = \mathbf{A}_0\mathbf{W}_0\mathbf{B}_0^T \tag{85}$$

$$= (140 \times 32) \cdot (32 \times 64) \cdot (64 \times 64) \tag{86}$$

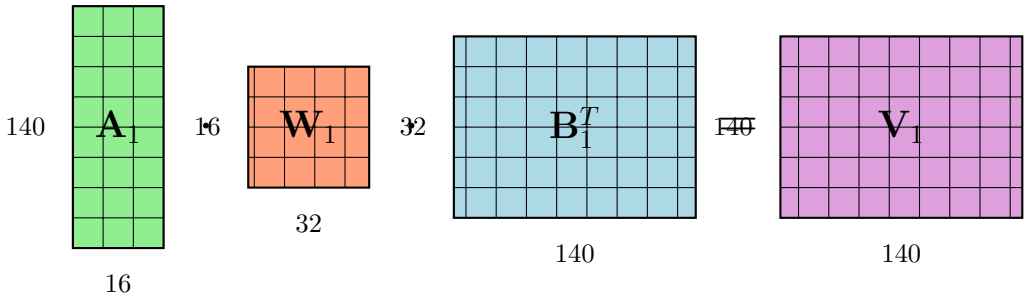
$$= (140 \times 64) \quad \checkmark \tag{87}$$

Note: Hidden layer expanded from 32 to 140, input (64) preserved.

G.4 Feed Layer 1 Transformation

$$\mathbf{V}_1 = \mathbf{A}_1 \mathbf{W}_1 \mathbf{B}_1^T \tag{88}$$

Original: (16×32), AWB: (140×140)



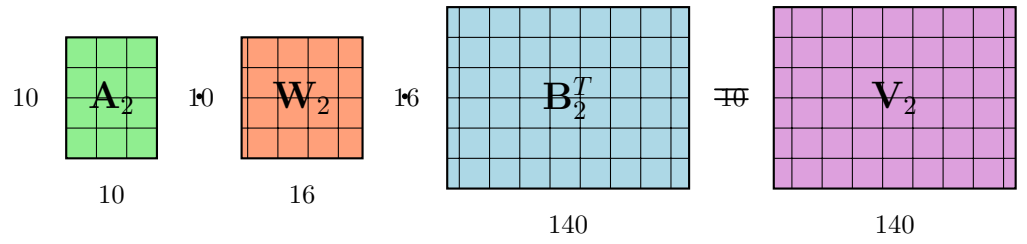
Dimension verification:

$$\begin{aligned} \mathbf{V}_1 &= \mathbf{A}_1 \mathbf{W}_1 \mathbf{B}_1^T && (89)\\ &= (140 \times 16) \cdot (16 \times 32) \cdot (32 \times 140) && (90)\\ &= (140 \times 140) \quad \checkmark && (91) \end{aligned}$$

G.5 Feed Layer 2 Transformation

$$\mathbf{V}_2 = \mathbf{A}_2 \mathbf{W}_2 \mathbf{B}_2^T \tag{92}$$

Original: (10×16), AWB: (10×140)



Dimension verification:

$$\begin{aligned} \mathbf{V}_2 &= \mathbf{A}_2 \mathbf{W}_2 \mathbf{B}_2^T && (93)\\ &= (10 \times 10) \cdot (10 \times 16) \cdot (16 \times 140) && (94)\\ &= (10 \times 140) \quad \checkmark && (95) \end{aligned}$$

Note: Output dimension (10) preserved, hidden layer expanded from 16 to 140.

H Summary and Comparison

H.1 Dimension Preservation Principle

Across all architectures, the AWB transformation follows the invariant:

Input and output dimensions preserved, hidden layers expanded

Architecture	Input Dim	Output Dim	Hidden Expansion
MLP	$64 \to 64$	$10 \to 10$	$128 \to 256$
CNN3D (feed)	$2304 \to 1600^*$	$10 \to 10$	$256 \to 512$
CNN (feed)	$1728 \to 1875^*$	$10 \to 10$	$64 \to 700$
GCN (gcn)	$5 \to 5$	$64 \to 64$	N/A
GCN (feed)	$64 \to 64$	$10 \to 10$	$32, 16 \to 140, 140$

Table 3: Dimension preservation across architectures. *Input changes due to convolutional filter expansion.

H.2 Matrix Dimension Formulas

For a layer with original weight $\mathbf{W} \in \mathbb{R}^{m_{\text{old}} \times n_{\text{old}}}$ transforming to $\mathbf{V} \in \mathbb{R}^{m_{\text{new}} \times n_{\text{new}}}$:

$$\mathbf{A} \in \mathbb{R}^{m_{\text{new}} \times m_{\text{old}}} \quad \text{(output transformation)} \tag{96}$$

$$\mathbf{B} \in \mathbb{R}^{n_{\text{new}} \times n_{\text{old}}} \quad \text{(input transformation)} \tag{97}$$

$$\mathbf{V} = \mathbf{AWB}^T \in \mathbb{R}^{m_{\text{new}} \times n_{\text{new}}} \tag{98}$$

H.3 Special Cases

Identity-like transformations:

- When dimensions are preserved: $\mathbf{A} \in \mathbb{R}^{k \times k}$ becomes identity-like
- Example: MLP Layer 0 input ($64 \to 64$), $\mathbf{B}_0 \in \mathbb{R}^{64 \times 64}$

Convolutional filters:

- 2D spatial transformation: $3 \times 3 \to 5 \times 5$
- Applied per-channel: (i, c) for output channel i, input channel c

I Conclusion

The AWB transformation $\mathbf{V} = \mathbf{AWB}^T$ enables seamless architecture morphing while preserving the fundamental input-output mapping of the network. The key insight is that transformation matrices $\mathbf{A}$ and $\mathbf{B}$ must be carefully dimensioned to:

1. Preserve input feature dimensions (first layer)
2. Preserve output class dimensions (last layer)
3. Allow flexible expansion of hidden layer dimensions

This framework applies uniformly across MLP, CNN, CNN3D, and GCN architectures, with special handling for convolutional filters and graph convolutions.

J Standard CL Implementation

This appendix provides comprehensive implementation details for the Hamiltonian Continual Learning (HCL) framework with Adaptive Weight Basis (AWB). We present the complete algorithmic pipeline that integrates all heuristics and techniques used in the standard implementation.

J.1 Overview

The standard CL implementation consists of:

- **Task 0**: Standard Hamiltonian-based training establishing the initial model
- **Tasks** $t \geq 1$: Full AWB pipeline with adaptive architecture morphing
- **Core Heuristics**: Task warmup, adaptive learning rates, adaptive gradient weights, balanced experience replay, and gradient normalization

All algorithms use the notation from the main text: $\mathbf{w}(t)$ denotes weights at task t, $\psi(t)$ denotes architecture, $\boldsymbol{\mathcal{X}}(t)$ denotes task data, $\hat{f}(\mathbf{w}, \psi)$ denotes the neural network, $J(\mathbf{w}(t), \psi(t), \boldsymbol{\mathcal{X}}(t))$ denotes forgetting loss, and $V(t, \mathbf{w}(t))$ denotes the cumulative loss (value function).

J.2 Main Algorithm

Algorithm 3: Complete Hamiltonian Continual Learning Pipeline

Input: Initial weights $\mathbf{w}(0)$, architecture $\psi(0)$, sequence of tasks $\{\boldsymbol{\mathcal{X}}(0), \boldsymbol{\mathcal{X}}(1), \ldots, \boldsymbol{\mathcal{X}}(T)\}$
Output: Final weights $\mathbf{w}(T)$ and architecture $\psi(T)$
Data: Hyperparameters: η (learning rate), $[\alpha, \beta, \gamma]$ (gradient weights), N_{epochs}
Initialize: Experience buffer $\mathcal{E} \leftarrow \emptyset$, optimizer state $\mathcal{O}$, records $\mathcal{R}$;
Set default gradient weights: $[\alpha_0, \beta_0, \gamma_0] = [0.01, 0.98, 0.1]$ **?**;
Set default learning rate: $\eta_0 = 10^{-4}$ **?**;
for $t = 0$ **to** T **do**
  `// Task Transition with Warmup (Algorithm 5)`
  if $t > 0$ ***and*** *warmup enabled* **then**
    $\eta_{\text{warmup}} \leftarrow 0.1 \cdot \eta$;
    $[\alpha, \beta, \gamma] \leftarrow [1.0, 0.0, 0.0]$ `// Current task only`
    for $e = 1$ **to** $N_{warmup} = 5$ **do**
      $\mathbf{w}(t) \leftarrow$ `train_step`$(\mathbf{w}(t), \psi(t), \boldsymbol{\mathcal{X}}(t), \eta_{\text{warmup}}, [\alpha, \beta, \gamma])$;
    end
  end
  `// Compute Adaptive Hyperparameters (Algorithm 6)`
  if $t > 0$ **then**
    $r_{\text{loss}} \leftarrow J(\mathbf{w}(t), \psi(t), \boldsymbol{\mathcal{X}}(t)) / J(\mathbf{w}(t-1), \psi(t-1), \boldsymbol{\mathcal{X}}(t-1))$ **?**;
    $\eta_{\min} \leftarrow \min(10^{-6}, 0.1 \cdot \eta_0 / r_{\text{loss}})$ `// Adaptive LR minimum`
    $w_{\text{curr}} \leftarrow \min(1.0, r_{\text{loss}})$ `// Current task weight`
    $w_{\text{exp}} \leftarrow 1.0 - w_{\text{curr}}$ `// Experience weight`
    $[\alpha, \beta, \gamma] \leftarrow [w_{\text{curr}} \cdot 0.01, w_{\text{exp}} \cdot 0.98, 0.1]$ **?**;
  end
  if $t = 0$ **then**
    `// Task 0: Standard Hamiltonian Training`
    for $e = 1$ **to** N_{epochs} **do**
      `// Get batch from current task`
      $\mathcal{B}_{\text{curr}} \leftarrow$ `sample`$(\boldsymbol{\mathcal{X}}(t))$;
      `// Compute Hamiltonian Gradient (Algorithm 4)`
      $\nabla_{\mathbf{w}} H \leftarrow \alpha \cdot \nabla_{\mathbf{w}} \ell(\hat{f}(\mathbf{w}(t), \psi(t)), \mathcal{B}_{\text{curr}})$;
      `// Apply gradient clipping`
      $\|\nabla_{\mathbf{w}} H\|_2 > 1.0 : \nabla_{\mathbf{w}} H \leftarrow \nabla_{\mathbf{w}} H / \|\nabla_{\mathbf{w}} H\|_2$ **?**;
      `// Update weights`
      $\mathbf{w}(t) \leftarrow$ `optimizer_step`$(\mathbf{w}(t), \nabla_{\mathbf{w}} H, \eta, \mathcal{O})$;
      `// Apply learning rate schedule`
      $\eta \leftarrow$ `lr_schedule`$(\eta, e, N_{\text{epochs}}, \eta_{\min})$ **?**;
    end
    `// Add task data to experience buffer`
    $\mathcal{E} \leftarrow \mathcal{E} \cup \{\text{samples from } \boldsymbol{\mathcal{X}}(t)\}$;
  else
    `// Tasks` $t \geq 1$`: AWB Pipeline`
    `// STEP 1: Preliminary Training`
    $J_{\text{prev}} \leftarrow J(\mathbf{w}(t), \psi(t), \boldsymbol{\mathcal{X}}(t))$;
    for $e = 1$ **to** $N_{prelim} = 100$ **do**
      `// Balanced Experience Replay (Algorithm 7)`
      $\mathcal{B}_{\text{curr}} \leftarrow$ `sample`$(\boldsymbol{\mathcal{X}}(t))$;
      $\mathcal{B}_{\text{exp}} \leftarrow$ `balanced_sample`$(\mathcal{E}, t)$ `// 10% recent, 80% older, 10% random`
      `// Compute full Hamiltonian gradient`
      $\nabla_{\mathbf{w}} V \leftarrow \nabla_{\mathbf{w}} \ell(\hat{f}(\mathbf{w}(t), \psi(t)), \mathcal{B}_{\text{exp}})$ **?**;
      $\nabla_{\mathbf{w}} \delta V \leftarrow$ `compute_dV`$(\mathbf{w}(t), \psi(t), \mathcal{B}_{\text{curr}}, \mathcal{B}_{\text{exp}})$ **?**;
      $\nabla_{\mathbf{w}} H \leftarrow \alpha \cdot \nabla_{\mathbf{w}} \ell_{\text{curr}} + \beta \cdot \nabla_{\mathbf{w}} V + \gamma \cdot \nabla_{\mathbf{w}} \delta V$;
      `// Normalize dV contribution`
      $\text{dV\_norm} \leftarrow \|\nabla_{\mathbf{w}} \delta V\|_2 / (t+1)$;
      $\nabla_{\mathbf{w}} H \leftarrow \alpha \cdot \nabla_{\mathbf{w}} \ell_{\text{curr}} + \beta \cdot \nabla_{\mathbf{w}} V + \gamma \cdot \text{dV\_norm}$;
      `// Gradient clipping and update`
      $\|\nabla_{\mathbf{w}} H\|_2 > 1.0 : \nabla_{\mathbf{w}} H \leftarrow \nabla_{\mathbf{w}} H / \|\nabla_{\mathbf{w}} H\|_2$

J.3 Supporting Algorithms

J.3.1 Hamiltonian Gradient Computation

Algorithm 4: Hamiltonian Gradient Computation

Input: Weights $\mathbf{w}$, architecture ψ, current batch $\mathcal{B}_{\text{curr}}$, experience batch $\mathcal{B}_{\text{exp}}$, gradient weights $[\alpha, \beta, \gamma]$
Output: Hamiltonian gradient $\nabla_{\mathbf{w}} H$

`// Current task gradient`
$\nabla_{\mathbf{w}} \ell_{\text{curr}} \leftarrow \nabla_{\mathbf{w}} \ell(\hat{f}(\mathbf{w}, \psi), \mathcal{B}_{\text{curr}})$;
`// Experience replay gradient (value function)`
$\nabla_{\mathbf{w}} V \leftarrow \nabla_{\mathbf{w}} \ell(\hat{f}(\mathbf{w}, \psi), \mathcal{B}_{\text{exp}})$;
`// Regularization term: sensitivity to perturbations`
$\sigma_x^2 \leftarrow 10^{-4}$, $\sigma_{\mathbf{w}}^2 \leftarrow 10^{-8}$ `// Perturbation variances ?`
`// Input perturbations`
$V_0 \leftarrow \ell(\hat{f}(\mathbf{w}, \psi), \mathcal{B}_{\text{exp}})$;
for $k = 1$ **to** 5 **do**
  $\mathcal{B}_{\text{exp}}^{(k)} \leftarrow \mathcal{B}_{\text{exp}} + \epsilon_x^{(k)}$, where $\epsilon_x^{(k)} \sim \mathcal{N}(0, \sigma_x^2 I)$;
  $V_k \leftarrow \ell(\hat{f}(\mathbf{w}, \psi), \mathcal{B}_{\text{exp}}^{(k)})$;
end
$\nabla_x V \leftarrow \frac{1}{5} \sum_{k=1}^{5} (V_k - V_0)/\sigma_x$ `// Finite difference approximation`
`// Parameter perturbations`
for $k = 1$ **to** 5 **do**
  $\epsilon_{\mathbf{w}}^{(k)} \sim \mathcal{N}(0, \sigma_{\mathbf{w}}^2 I)$;
  $\mathbf{w}^{(k)} \leftarrow \mathbf{w} + \epsilon_{\mathbf{w}}^{(k)}$;
  $V_k^{\mathbf{w}} \leftarrow \ell(\hat{f}(\mathbf{w}^{(k)}, \psi), \mathcal{B}_{\text{exp}})$;
end
$\nabla_{\mathbf{w}} V_{\text{perturb}} \leftarrow \frac{1}{5} \sum_{k=1}^{5} (V_k^{\mathbf{w}} - V_0)/\sigma_{\mathbf{w}}$;
$\nabla_{\mathbf{w}} \delta V \leftarrow \nabla_x V + \nabla_{\mathbf{w}} V_{\text{perturb}}$ `// Total regularization`
`// Combine all components`
$\nabla_{\mathbf{w}} H \leftarrow \alpha \cdot \nabla_{\mathbf{w}} \ell_{\text{curr}} + \beta \cdot \nabla_{\mathbf{w}} V + \gamma \cdot \nabla_{\mathbf{w}} \delta V$ **?**;
Return $\nabla_{\mathbf{w}} H$;

J.3.2 Task Transition with Warmup

Algorithm 5: Task Transition with Warmup

Input: Weights $\mathbf{w}(t)$, architecture $\psi(t)$, new task $\boldsymbol{\mathcal{X}}(t)$, base LR η_0
Output: Warmed-up weights $\mathbf{w}(t)$

`// Reduce learning rate for warmup`
$\eta_{\text{warmup}} \leftarrow 0.1 \cdot \eta_0$ **?**;
`// Focus only on current task (disable experience replay)`
$[\alpha, \beta, \gamma] \leftarrow [1.0, 0.0, 0.0]$;
for $e = 1$ **to** $N_{warmup} = 5$ **do**
  $\mathcal{B}_{\text{curr}} \leftarrow$ `sample`$(\boldsymbol{\mathcal{X}}(t))$;
  $\nabla_{\mathbf{w}} \ell_{\text{curr}} \leftarrow \nabla_{\mathbf{w}} \ell(\hat{f}(\mathbf{w}(t), \psi(t)), \mathcal{B}_{\text{curr}})$;
  $\mathbf{w}(t) \leftarrow$ `optimizer_step`$(\mathbf{w}(t), \nabla_{\mathbf{w}} \ell_{\text{curr}}, \eta_{\text{warmup}}, \mathcal{O})$;
end
Return $\mathbf{w}(t)$;

J.3.3 Adaptive Hyperparameter Computation

Algorithm 6: Adaptive Learning Rate and Gradient Weights

Input: Current loss J_{curr}, previous loss J_{prev}, base LR η_0, base weights $[\alpha_0, \beta_0, \gamma_0]$
Output: Adapted LR $\eta_{\min}$ and gradient weights $[\alpha, \beta, \gamma]$
// Compute loss ratio
$r_{\text{loss}} \leftarrow J_{\text{curr}}/J_{\text{prev}}$?;
// Adaptive learning rate minimum
$\eta_{\min} \leftarrow \min(10^{-6}, 0.1 \cdot \eta_0/r_{\text{loss}})$ // Lower bound increases with difficulty
// Adaptive gradient weights based on task difficulty
$w_{\text{curr}} \leftarrow \min(1.0, r_{\text{loss}})$ // Weight for current task
$w_{\text{exp}} \leftarrow 1.0 - w_{\text{curr}}$ // Weight for experience
$\alpha \leftarrow w_{\text{curr}} \cdot \alpha_0$ // Current task gradient weight
$\beta \leftarrow w_{\text{exp}} \cdot \beta_0$ // Experience gradient weight ?
$\gamma \leftarrow \gamma_0$ // Regularization weight (fixed)
Return $\eta_{\min}, [\alpha, \beta, \gamma]$;

J.3.4 Balanced Experience Replay

Algorithm 7: Balanced Experience Replay Buffer Sampling

Input: Experience buffer $\mathcal{E}$, current task ID t, batch size B
Output: Experience batch $\mathcal{B}_{\text{exp}}$
// Compute sampling quotas
$n_{\text{recent}} \leftarrow \lfloor 0.1 \cdot B \rfloor$ // 10% from recent task
$n_{\text{older}} \leftarrow \lfloor 0.8 \cdot B \rfloor$ // 80% from older tasks
$n_{\text{random}} \leftarrow B - n_{\text{recent}} - n_{\text{older}}$ // 10% uniform random
// Sample from recent task ($t-1$)
if $t > 0$ **then**
  $\mathcal{S}_{\text{recent}} \leftarrow \texttt{sample}(\mathcal{E}_{t-1}, n_{\text{recent}})$;
else
  $\mathcal{S}_{\text{recent}} \leftarrow \emptyset$;
end
// Sample from older tasks (0 to $t-2$) uniformly
if $t > 1$ **then**
  $n_{\text{per\_task}} \leftarrow \lceil n_{\text{older}}/(t-1) \rceil$;
  $\mathcal{S}_{\text{older}} \leftarrow \bigcup_{i=0}^{t-2} \texttt{sample}(\mathcal{E}_i, n_{\text{per\_task}})$;
else
  $\mathcal{S}_{\text{older}} \leftarrow \emptyset$;
end
// Sample uniformly from all tasks
$\mathcal{S}_{\text{random}} \leftarrow \texttt{sample}(\mathcal{E}, n_{\text{random}})$?;
// Combine samples
$\mathcal{B}_{\text{exp}} \leftarrow \mathcal{S}_{\text{recent}} \cup \mathcal{S}_{\text{older}} \cup \mathcal{S}_{\text{random}}$;
Return $\mathcal{B}_{\text{exp}}$;

J.3.5 AWB Architecture Change Decision

Algorithm 8: AWB Architecture Change Decision

Input: Loss before preliminary training J_{prev}, loss after J_{new}
Output: Decision: change architecture (True/False)
`// Compute loss ratio and change`
$r_{\text{loss}} \leftarrow J_{\text{new}}/J_{\text{prev}}$;
$\Delta J \leftarrow J_{\text{new}} - J_{\text{prev}}$;
`// Architecture change criteria`
$\theta_{\text{high}} \leftarrow 0.9$ `// High loss ratio threshold`
if $(r_{loss} > \theta_{high})$ ***and*** $(\Delta J > 0)$ **then**
  Return True `// Loss increased significantly, change architecture`
else
  Return False `// Continue with current architecture`
end

J.3.6 Architecture Search

Algorithm 9: NDDS Architecture Search

Input: Current architecture $\psi(t)$, task data $\boldsymbol{\mathcal{X}}(t)$, weights $\mathbf{w}(t)$
Output: Optimal architecture $\psi^*(t)$
`// Extract current dimensions`
$\{d_1, d_2, \ldots, d_L\} \leftarrow$ `extract_dims`$(\psi(t))$ `// Layer dimensions`
`// Define search space`
$\mathcal{S} \leftarrow \{d_i \pm k \cdot s : i \in \{1, \ldots, L\}, k \in \{1, 2, 3\}\}$
where $s = 16$ is step size;
`// Neighborhood Directional Direct Search ?`
$\psi^* \leftarrow \psi(t)$, $J_{\text{best}} \leftarrow \infty$;
foreach *candidate architecture* $\psi' \in \mathcal{S}$ **do**
  `// Evaluate on validation subset`
  $\mathbf{w}' \leftarrow$ `init_weights`(ψ') `// Transfer from` $\mathbf{w}(t)$ `if possible`
  $J' \leftarrow$ `quick_eval`$(\mathbf{w}', \psi', \boldsymbol{\mathcal{X}}_{\text{val}}(t))$ `// 50 samples, 10 epochs`
  if $J' < J_{best}$ **then**
    $J_{\text{best}} \leftarrow J'$;
    $\psi^* \leftarrow \psi'$;
  end
end
Return ψ^*;

J.4 Hyperparameter Reference

Table 4 summarizes all default hyperparameters used in the standard CL implementation.

J.5 Implementation Notes

J.5.1 Gradient Normalization

The $\nabla_{\mathbf{w}} \delta V$ component is normalized by the number of tasks to prevent its magnitude from dominating as more tasks are learned:

$$\text{dV\_norm} = \frac{\|\nabla_{\mathbf{w}} \delta V\|_2}{t+1} \tag{99}$$

This ensures balanced contributions from all gradient components throughout the learning process.

Table 4: Default Hyperparameters for Standard CL Implementation

Parameter	Default Value	Description
Gradient Computation		
α	0.01	Current task gradient weight
β	0.98	Experience replay gradient weight
γ	0.1	Regularization gradient weight
σ_x^2	10^{-4}	Input perturbation variance
$\sigma_{\mathbf{w}}^2$	10^{-8}	Parameter perturbation variance
Optimization		
η_0	10^{-4}	Base learning rate
$\eta_{\min}$	10^{-6}	Minimum learning rate
Optimizer	Adam	Default optimizer
Gradient clip	1.0	Maximum gradient norm
Task Warmup		
N_{warmup}	5 epochs	Warmup duration
LR factor	0.1	Warmup learning rate multiplier
Experience Replay		
Buffer size	200,000	Maximum samples
Recent quota	10%	From task $t-1$
Older quota	80%	From tasks 0 to $t-2$
Random quota	10%	Uniform random
AWB Pipeline		
N_{prelim}	100 epochs	Preliminary training
N_{AB}	50 epochs	A/B matrix training
θ_{high}	0.9	Architecture change threshold
ϵ_{AB}	$0.01 \cdot (1+0.1t)^{-1}$	A/B convergence threshold
Search step	16	Dimension search step size

J.5.2 Learning Rate Schedules

The framework supports multiple LR schedules **?**:

- **Constant**: $\eta(e) = \eta_0$
- **Step**: $\eta(e) = \eta_0 \cdot 0.1^{\lfloor e/100 \rfloor}$
- **Exponential**: $\eta(e) = \eta_0 \cdot \exp(-0.01 \cdot e)$
- **Cosine**: $\eta(e) = \eta_{\min} + \frac{1}{2}(\eta_0 - \eta_{\min})(1 + \cos(\pi e / N_{\text{epochs}}))$
- **Linear**: $\eta(e) = \eta_0 \cdot (1 - e/N_{\text{epochs}})$

J.5.3 JAX Implementation

All gradient computations are JIT-compiled using JAX for performance:

- Separate functions for regression vs. classification
- Separate functions for standard vs. AWB mode
- Separate functions for vector vs. graph problems
- 8 JIT-compiled variants total for different problem types

J.5.4 Model Partitioning

Equinox models are partitioned into trainable and static components:

- **Standard training**: All weights trainable
- **A/B training**: Freeze $\mathbf{w}$, train A, B
- **V training**: Freeze A, B, train $V = \mathrm{A} \cdot \mathbf{w} \cdot \mathrm{B}^T$