Partial Hypernetworks for Continual Learning

Hamed Hemati[1], Vincenzo Lomonaco[2], Davide Bacciu[2], Damian Borth[1]

hamed.hemati@unisg.ch, vincenzo.lomonaco@unipi.it, davide.bacciu@unipi.it, damian.borth@unisg.ch

[1]University of St. Gallen, Switzerland [2] University of Pisa, Italy

Abstract

Hypernetworks mitigate forgetting in continual learning (CL) by generating task-dependent weights and penalizing weight changes at a meta-model level. Unfortunately, generating all weights is not only computationally expensive for larger architectures, but also, it is not well understood whether generating all model weights is necessary. Inspired by latent replay methods in CL, we propose partial weight generation for the final layers of a model using hypernetworks while freezing the initial layers. With this objective, we first answer the question of how many layers can be frozen without compromising the final performance. Through several experiments, we empirically show that the number of layers that can be frozen is proportional to the distributional similarity in the CL stream. Then, to demonstrate the effectiveness of hypernetworks, we show that noisy streams can significantly impact the performance of latent replay methods, leading to increased forgetting when features from noisy experiences are replayed with old samples. In contrast, partial hypernetworks are more robust to noise by maintaining accuracy on previous experiences. Finally, we conduct experiments on the split CIFAR-100 and TinyImagenet benchmarks and compare different versions of partial hypernetworks to latent replay methods. We conclude that partial weight generation using hypernetworks is a promising solution to the problem of forgetting in neural networks. It can provide an effective balance between computation and final test accuracy in CL streams.

1 Introduction

One of the core assumptions in statistical learning is that the data samples from a given dataset are independent and identically distributed (IID) Hastie et al. (2009). This assumption requires the model to be trained and evaluated on a fixed data distribution. However, there are different cases in which this assumption can be violated due to changes in the data distribution, either in training or in the test phase. Such cases include multi-task learning, adversarial examples, and continual learning Murphy (2022).

Continual learning focuses on a model's ability to learn from new experiences over time Thrun & Mitchell (1995) when exposed to a data stream. This capability is particularly useful in real-world applications across different domains, such as image understanding Parisi et al. (2019); De Lange et al. (2021), natural language processing Biesialska et al. (2020), reinforcement learning Khetarpal et al. (2020), and robotics Lesort et al. (2020). However, there are multiple challenges to learning from a stream of experiences. One such challenge is catastrophic forgetting (CF) French (1999), which occurs when information from older experiences is partially or completely replaced by information from newer experiences. This can be problematic when models are deployed for real-world scenarios, as the model must adapt its knowledge to new situations while preserving its previously learned information.

Rehearsal-based methods are a well-known category of strategies that are highly regarded for their simplicity and efficiency in mitigating CF by replaying exemplars from previously encountered experiences. These exemplars may take on a variety of forms, including raw samples from the dataset Rolnick et al. (2019), synthesized samples generated via a generative model Shin et al. (2017); Graffieti et al. (2022), or in latent replay (LR) form where features are obtained from a frozen feature extractor Ostapenko et al. (2022); Demosthenous & Vassiliades (2021). In LR, the number of frozen layers after the pre-training phase can significantly impact the model's ability to learn from subsequent experiences. For instance, freezing all model layers, except for the classification layer, is reasonable only when there are close affinities between upcoming experiences and the one used for pre-training, as the feature extractor directly influences the quality of features.

Given these points, critical questions need to be addressed in LR methods with frozen feature extractors. The first question is *How many layers can we freeze without losing generalizability?* And the second question is *How sensi-*

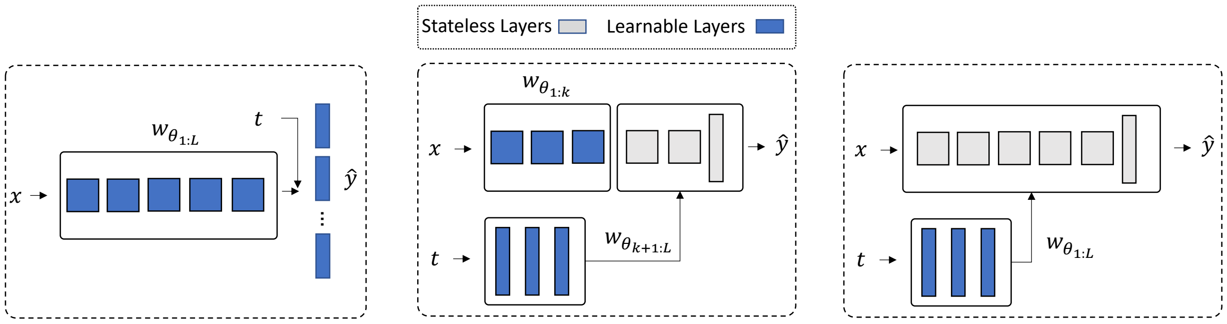


Figure 1: Left: a regular network with multiple heads. Middle: a partial hypernetwork that generates the weights of the final layers of the main model. Right: a hypernetwork that generates all weights of the main model.

tive are these approaches to distributional changes? The first question can be partly addressed by allocating larger capacity to the model layers succeeding the frozen feature extractor by finding a sound trade-off between the number of frozen and non-frozen layers. The latter, however, requires a more careful examination of how replaying features extracted from a fixed feature extractor affects the model's learning across different distributions in terms of negative interference, an issue also studied by Wang et al. (2021).

One possible approach to address multi-task learning problems with high negative interference between tasks is to use Hypernetworks (HNs) Ha et al. (2016). HNs are meta-models that produce the weights of another network, commonly known as the "main model". The weights can be generated in response to different types of signals, such as the task ID in a multi-task learning dataset or the input itself to create input-specific weights. HNs offer several advantages, including model compression Nguyen et al. (2021), neural architecture search Brock et al. (2017), and continual learning Von Oswald et al. (2019). By capturing the shared structure between tasks, HNs can improve generalization to new tasks Beck et al. (2022) through adaptive weight generation. In continual learning, HNs have been demonstrated to mitigate forgetting significantly Von Oswald et al. (2019), although training can be computationally expensive Cubélier et al. (2022). However, HNs can be used in a more efficient way. For example, Üstün et al. (2022) proposed generating only the weights of adaptive layers in a neural network. This approach can potentially reduce the computational costs associated with HNs while still leveraging their benefits.

We propose to use partial hypernetworks to generate the weights of non-frozen layers in a model as an alternative to replaying stored features. This idea is inspired by recent work on LR with frozen layers and motivated by the observation that lower layer representations remain stable after the first experience Ramasesh et al. (2020). Using HNs for the final layers provides advantages over regular LR, and standard HN approaches. First, we avoid storing exemplars and instead perform replay at the meta-model level by keeping the converged version of the hypernetwork from the previous step, which is typically smaller than the main model. Second, we avoid generating the weights of layers shared among experiences, which can speed up model training while achieving comparable performance to regular HNs. We comprehensively analyze different hypernetwork architectures and evaluate their performance on various benchmark datasets to compare these models with their LR and standard HN counterparts. Below, we summarize our contributions:

- We propose a more efficient solution for hypernetworks by combining the concept of a frozen feature extractor with hypernetworks.
- We demonstrate that generating partial weights using hypernetworks can achieve comparable performance under specific conditions.
- We offer insights into when these models can be a superior alternative to conventional latent replay techniques.

2 Problem Definition

In a standard supervised classification problem using neural networks, we assume we have a neural network $\mathbf{f}_\theta(x)$ with learnable parameters θ. Given a dataset $\mathbf{D} = \{(x, y); x \in \mathbf{X}, y \in \mathbf{Y}\}$ with underlying distribution $\mathcal{D}$ where $\mathbf{X} \subseteq \mathbb{R}^d$ and $\mathbf{Y} = \{1, 2, \ldots C\}$ represent the input features with dimensionality d and output labels in the category space, respectively; the objective is to learn a mapping $\mathbf{f}_\theta : \mathbf{X} \rightarrow \mathbf{Y}$ that has a low empirical risk over the dataset samples. In other words, we seek to minimize the difference between the mappings and ground-truth samples by finding $\theta^* =$

$\arg\min_\theta L$, where $L = \sum_{i=0}^{N} CE(\mathbf{f}_\theta(x), y)$ denotes the empirical risk, and CE is the function that calculates the cross-entropy loss, with samples drawn from $\mathcal{D}$ in an IID fashion. We obtain a solution with a predetermined error threshold using the stochastic gradient descent (SGD) algorithm.

Continual Learning in contrast to the standard supervised learning, is a setting in which the entire dataset $\mathbf{D}$ is not available at once. Instead, data becomes available over time in the form of a data stream. Specifically, the model is exposed to a sequence of experiences $\mathcal{S} = \{\mathbf{e}_1, \mathbf{e}_2, ..., \mathbf{e}_N\}$, where each $\mathbf{e}_i, 0 < i < N$ represents an individual experience that contains a train and test set, i.e., $\mathbf{e}_i = \{\mathbf{D}_{tr}^{(i)}, \mathbf{D}_{te}^{(i)}, t^{(i)}\}$ with $\mathbf{D}_{tr}^{*}$, $\mathbf{D}_{te}^{*}$ and $t^{(i)}$ being the train set, test set and the task id of the experience. The datasets in each experience can be drawn from the same distribution or have different distributions.

A *strategy* refers to an algorithm that updates the model $\mathbf{f}_\theta$ over a sequence of experiences. When a strategy goes through each experience, it does so one after the other and does not revisit any previously observed experience. Once the strategy observes the N^{th} experience $\mathbf{e}_N$, it obtains the corresponding converged parameters θ_N^*. After each experience, the model is evaluated by calculating the Average Classification Accuracy (ACA) over the test set of all the experiences observed so far, i.e., ACA $= \sum_{i=1}^{N} \text{Acc}(\mathbf{f}_{\theta_T^*}, \mathbf{D}_{te}^{(i)})$ with $\text{Acc}(\mathbf{f}, \mathbf{D})$ being the accuracy of a function $\mathbf{f}$ over dataset $\mathbf{D}$. Strategies are typically compared in terms of their ability to preserve knowledge, which refers to the degree to which the model's performance is affected after going through all experiences in the stream.

The type of *scenario* in CL determines what kind of information a new experience provides and how the input and output domains change over time. The most commonly encountered scenarios are domain-incremental, task-incremental, and class-incremental Van de Ven & Tolias (2019). In the domain-incremental scenario, the category space remains constant for all experiences, while only the "domain" of the mapping changes. In both task-incremental and class-incremental scenarios, the category space changes as new classes are introduced. However, in the task-incremental scenario, the experience indicator is given during test time, while in the class-incremental scenario, the model is not presented with any form of task indicator at test time. It is possible to formulate the class-incremental scenario as a task-incremental learning problem with an additional task inference step as theoretically formulated in Kim et al. (2022). In this case, the mapping $\mathbf{f}$ becomes $\mathbf{f} : \mathbf{X} \times \mathbf{T} \rightarrow \mathbf{Y}$ where in the case of class-incremental, $\mathbf{T}$ has to be inferred. In our experimental setting, similar to Von Oswald et al. (2019), we assume that the task ID is either given or inferred, which converts our problem into a task-incremental learning problem.

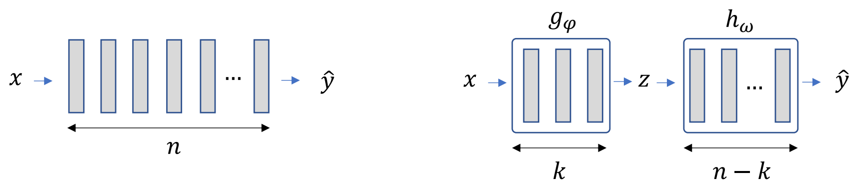


Figure 2: Decomposition of a neural network with n layers into two parts with decomposition depth k.
.

Model Decomposition is an operation that converts a function $\mathbf{f}$ into a composite function $\mathbf{f}_P(\mathbf{f}_{P-1}(...(\mathbf{f}_1(x))))$, where P is the total number of consecutive chunks that result from splitting the neural network into P parts. By decomposing a function into simpler parts, we can gain insights into its behavior and use it to solve problems more flexibly. For example, we often decompose functions in calculus to apply techniques like the chain rule or integration by substitution. A common form of decomposition in neural networks is to break them down into two simpler neural networks. In the case of LR with frozen layers, we decompose neural networks into two parts, where the parameters in the first half's layers are frozen and used for feature extraction. In our problem setting, we perform depth-wise decomposition by always splitting a neural network based on a given depth. For depth-specific decomposition, we consider a model $\mathbf{f}_{\theta,l}$ with $l \geq 2$, where l indicates the number of layers, and we define a decomposition for depth k as below:

$$O_{dec}(\mathbf{f}_\theta, k) = \mathbf{f}_{\theta^{1:k}}(\mathbf{f}_{\theta^{k:n}}(x)) \tag{1}$$

O_{dec} denotes the decomposition operation , which is applied for depth k, and $\theta^{a:b}$ denotes the subset of parameters from consecutive layers a to b. In the case of LR, k is a hyper-parameter that should be set based on the model architecture, computational limitations, and the level of shared structure across tasks. An alternative notation to represent the model segments obtained after decomposition is $\mathbf{f}_{\theta^{1:k}} = \mathbf{g}_\phi$ and $\mathbf{f}_{\theta^{k:l}} = \mathbf{h}_\omega$. With this notation, the inference is performed by first computing $z = \mathbf{g}_\phi(x)$ and then using $\mathbf{h}_\omega$ to compute $\hat{y} = h_\omega(z)$. Figure 2 provides an example of such a decomposition.

Partial Weight Generation is a technique utilized for task-conditional prediction in multi-task or continual learning problems. The primary goal is to generate task-specific weights for particular layers of a model $\mathbf{f}$, and HNs are mainly employed for this purpose. The hypernetwork $\mathbf{H}_\Psi$ generates the parameters θ of model $\mathbf{f}$ before making prediction. When predicting input x, $\mathbf{H}_\Psi$ first generates the weights conditioned on an additional conditioning vector t, which can be the input x itself or a task-dependent indicator such as the task ID. This weight generation process can also be considered an adaptive approach for prediction since the predictor's weights are initially adapted in a generative way for the given task ID and then used for prediction. In our setting, we always decompose $\mathbf{f}$ into two parts, i.e. $\mathbf{h}_\omega(\mathbf{g}\phi(x))$, as shown in Equation 1 and make prediction as below:

$$w_\phi = \mathbf{H}_\psi(t) \tag{2}$$
$$z = \mathbf{g}_\phi(x; w_\phi) \tag{3}$$
$$\hat{y} = \mathbf{h}_\omega(z) \tag{4}$$

The layers corresponding to the first half are called learnable or *stateful* layers and the layers in the second parts whose weights are generated by the hypernetwork are referred to as *stateless* layers.

Continual Learning with Decomposed Models exploits a decomposed neural network $\mathbf{f}(\mathbf{x}) = \mathbf{h}_\omega(\mathbf{g}_\phi(x))$ with decomposition depth k to learn from a stream of experiences with minimal catastrophic forgetting. At each step of training, we are given a batch of data samples (x, y, t) which is then used to update $\mathbf{f}$. For our experimental setup, we assume that we freeze the parameters ϕ after the first experience in the stream. Regarding used strategies, we mainly consider two sets of methods: LR and partial HNs. The LR methods are facilitated with an additional buffer $\mathcal{B}$ with a limited capacity which stores features from previous samples extracted from $\mathbf{g}_\phi$. The details of the HN architecture are given in Appendix A.

3 Method

In this section, we describe our method that updates a partial HN over a stream of experiences. We split the method into two phases. In the first phase, the model is trained on the first experience until convergence, then the first part of the model is frozen. From the second experience, the model is updated using a look-ahead mechanism Zhang et al. (2019); Gupta et al. (2020); Von Oswald et al. (2019).

At each training step, we are given a batch of data samples $b = (x, y, t)$ where t is the experience ID. We assume $\theta = \{\phi, \omega\}$ constitute the parameters of model $\mathbf{f}$ and the hypernetwork is parameterized by ψ. We show the steps of our method in Algorithm 1 in Appendix D.

Experience e_1: In the first experience e_1, we train the model with regular SGD updates. Using task indicator t, we first infer the parameters ω of the stateless layers of the main model via the hypernetwork: $\omega = \mathbf{H}_\psi(t)$. After generating the stateless weights in the main model, we make a prediction on the batch inputs x to obtain $\hat{y} = \mathbf{h}(z = \mathbf{g}(x; \omega))$. The cross-entropy loss $\mathcal{L}_{CE} = CE(y, \hat{y})$ is used to compute the gradients of learnable (stateful) layers and the hypernetwork:

$$\phi \leftarrow \phi - \alpha * \frac{\partial \mathcal{L}_{CE}}{\partial \phi} \tag{5}$$
$$\psi \leftarrow \psi - \beta * \frac{\partial \mathcal{L}_{CE}}{\partial \psi} \tag{6}$$

where α and β are the learning rates for the main network and the hypernetwork, accordingly. The same process is repeated until convergence to obtain $H_\psi^{*(1)}$.

Experience $e_i, i \geq 2$: Before starting each new experience e_i, we first store a copy of the converged hypernetwork $\mathbf{H}_\psi^* \leftarrow \mathbf{H}_\psi^{*(i-1)}$ from the previous experience. Similar to Von Oswald et al. (2019), we use $\mathbf{H}_\psi^*$ to penalize the hypernetwork's outputs for previous experiences:

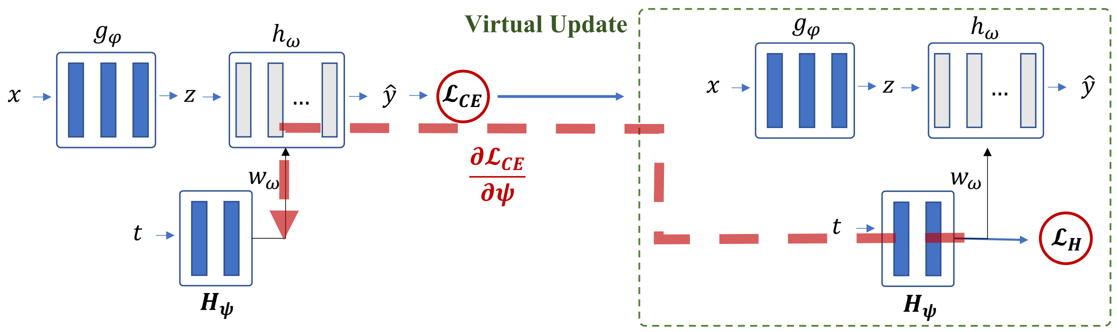


Figure 3: Illustrations of look-ahead gradient computation for the parameters of $\mathbf{H}$.

$$L_H = \sum_{j=1}^{i-1} \|\mathbf{H}^*_\psi(i) - \mathbf{H}_\psi(i)\|_2^2 \tag{7}$$

A naive way to compute the gradients for each step would be to add the regularization term to the cross-entropy loss directly and backpropagate through the network. This would be problematic on the current batch as it will result in inaccurate gradients. We show a visualization of this effect in Appendix B.1. To fruther explain this effect, we give an example of why this happens:

Let's assume the model is provided with batch $b = (x, y, t)$. In the first step of the experience, the regularization loss will be zero since H_ψ only changes after making an update on the current batch b. Therefore the gradients computed via $\mathcal{L}_H$ will always be one step behind, and the final gradient obtained for each step will be inaccurate. This may not have a big impact in streams with similar tasks, however in streams with big shifts in the distributions of the tasks, it can results in a noticeable difference. Simply put, we aim to find a way to incorporate the impact of the current update made via $\mathcal{L}_{CE}$ on the hypernetwork's outputs for previous tasks.

One solution is to use a different update method such as look-ahead to also obtain gradients for penalizing the changes of H_ψ from the first step of the experience. Look-ahead is an optimization method, where instead of *immediately* updating the model on a given batch of data, it first virtually updates the model for one or more steps, then computes the gradients in the final step of the virtual model, w.r.t. the original parameters. This way, we can compute more accurate gradient ψ by backpropagating through the optimization trajectory to measure the impact of the changes made by the new batch b on the parameters that affect the previous tasks. This process is shown in Figure 11.

Thus, the gradient calculation is done in two steps:

Step 1: Calculate loss $\mathcal{L}_{CE}$ on the current batch and compute gradients w.r.t. $\phi^{(0)} = \phi$ and $\psi^{(0)} = \psi$:

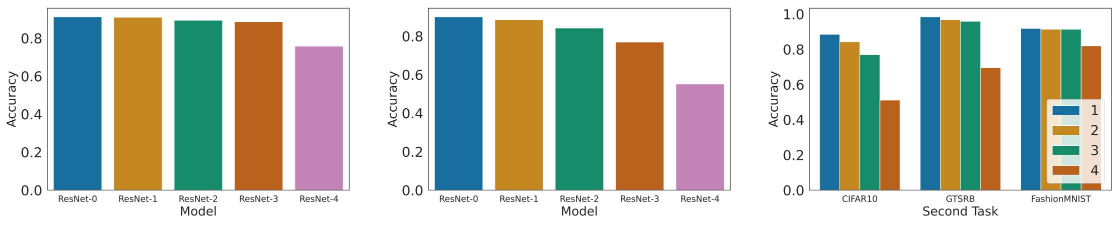
$$\nabla_\psi \mathbf{H}_1 = \frac{\partial \mathcal{L}_{CE}}{\partial \psi^{(0)}} \tag{8}$$

Use $\nabla_\psi \mathbf{H}_1$ to virtually update the model without changing the original model.

Step 2: Calculate loss $\mathcal{L}_\mathbf{H}$ on the virtual model to compute the gradients w.r.t. the parameters in the first step:

$$\nabla_\psi \mathbf{H}_2 = \frac{\partial \mathcal{L}_\mathbf{H}}{\partial \psi^{(0)}} = \frac{\partial J_\mathbf{H}(\psi^{(0)} - \beta \frac{\partial \mathcal{L}_{CE}}{\partial \psi^{(0)}})}{\partial \psi^{(0)}} = J'_\mathbf{H}(.) + \frac{\partial}{\partial \psi^{(0)}}(\psi^{(0)} - \alpha \frac{\partial \mathcal{L}_{CE}}{\partial \psi^{(0)}}) = J'_\mathbf{H}(.) - \alpha \frac{\partial^2 \mathcal{L}_{CE}}{\partial {\psi^{(0)}}^2} \tag{9}$$

where $J_\mathbf{H}$ and $J'_\mathbf{H}$ represent the loss function used to calculate $\mathcal{L}_H$ as in Equation 7. To obtain $\nabla_\psi \mathbf{H}_2$, we need to compute the second-order derivatives as shown in blue. In Appendix B.2, we show that even a first-order approximation achieves similar performance, thus in practice, we can use a first-order approximation.



(a) CIFAR-10 $\rightarrow$ CIFAR-100 (b) GTSRB $\rightarrow$ CIFAR-100 (c) SVHN $\rightarrow$ {CIFAR-10, GTSRB, FashionMNIST}. Numbers show freeze depth.

Figure 4: Test accuracy in the second experience for different set of streams.

Step 3: Finally, we need to compute the final gradients to update the main model. The gradients in the first step correspond to the changes that are useful for the current batch, and the gradients in the second step are used for the changes that can help to avoid forgetting the previous tasks. Therefore, we use the mean of the gradients:

$$\nabla_{\psi}\mathbf{H} = \frac{\nabla_{\psi}\mathbf{H}_1 + \nabla_{\psi}\mathbf{H}_2}{2} \tag{10}$$

$\nabla_{\psi}\mathbf{H}$ is used to update the parameters of the hypernetwork at each training step.

4 Experiments

In this section, we conduct experiments to demonstrate the efficacy of partial weight generation using HNs. First, we justify the approach by showing that under certain conditions, full weight generation is unnecessary. We then show that HNs are more robust to noises in the stream. Finally, we test partial HNs and LR on the Split-CIFAR100 and Split-TinyImagenet benchmarks as two standard test beds and compare them for different freezing depths. Throughout all experiments, we utilize slim ResNet-18 Lopez-Paz & Ranzato (2017) as the base model, which usually serves as a standard model for comparison in continual learning papers. For the hypernetwork, we employ a multi-layer perception with two hidden layers and separate weight generation heads for each layer of the main model. Additional details about the model architectures can be found in Appendix A.

The main model architecture consists of a convolutional layer, four residual blocks, and a classification layer. We adopt a block-wise freezing approach, wherein *the freeze depth* k implies that the first k blocks of the model are frozen including the initial convolutional layer. This allows us to create five distinct versions of the base model, denoted by ResNet-k where $k = \{0, 1, 2, 3, 4\}$ determines the number of blocks up to which the model is frozen. For instance, ResNet-0 indicates no frozen layers, whereas ResNet-4 implies that all layers except the classification layer are frozen. In the HN versions, the weights of the classification layer are generated by the hypernetwork. We implement all strategies using the Avalanche library Lomonaco et al. (2021). The codebase of our implementation is publicly available at https://github.com/HamedHemati/Partial-Hypernetworks-for-CL.git.

4.1 Analyzing the Effect of Frozen Layers When Learning New Tasks

Our investigation begins by asking the following question:

How many layers can be frozen without a significant reduction in the performance?

Obtaining an answer to this question provides a good starting point for applying partial weight generation via HNs. To this end, we design a small experiment using five different datasets: FashionMNIST Xiao et al. (2017), SVHN Netzer et al. (2011), GTSRB, CIFAR-10, and CIFAR-100 Krizhevsky et al. (2009). By leveraging the variations and complexities across these datasets, we can assess the extent to which frozen layers negatively impact the model's performance. For this purpose, we create two sets of CL streams based on distributional similarities between experiences. In the first set, each stream entails two experiences with five random classes in each experience drawn from different distributions. For example, a stream may consist of samples from the FashionMNIST dataset in the first experience and the CIFAR-10 dataset in the second experience. We want to measure the learning accuracy in the second experience by enforcing the model to use the frozen layers learned in the first experience from a different distribution with a

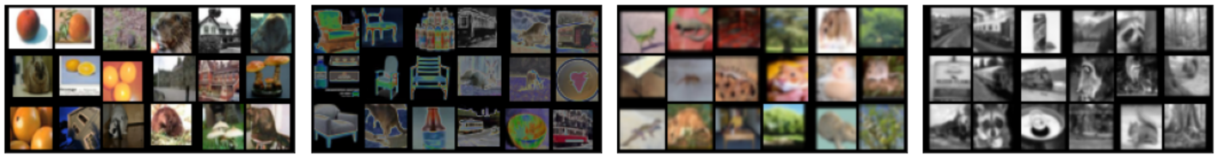


Figure 5: Random samples from a noisy stream consisting of 4 experiences. Each experience consists of five distinct classes.

varying number of frozen blocks. In the second set, we similarly create streams of length two with five random classes in each experience, but we sample from CIFAR-10 for the first experience and from CIFAR-100 for the second experience. Since both datasets come from similar distributions, we want to examine how distribution similarity between experiences affects the model for changing freeze depths.

In Figure 4 (a and b), we show that when the model is first trained on five random classes from the CIFAR-10 dataset, learning the next experience drawn from CIFAR-100 can work well for most versions of ResNet-k model as the accuracy drop is minimal. However, if we first train the model on the GTSRB dataset, learning a new experience from the CIFAR-100 dataset can be significantly affected depending on the model version, i.e., the number of frozen blocks. Moreover, in Figure 4 (c), we demonstrate that the complexity of subsequent experiences in a stream also plays an important role. For example, training a model on SVHN in the first experience and freezing the first three blocks doesn't cause a noticeable reduction in the performance when learning the FashionMNIST dataset since the new experience is simpler and doesn't require complex features. This suggests that the distribution variation in a stream is an important indicator in the transferability of features among experiences however, different factors such as freeze depth, the affinities between experiences, and the difficulty of the new experience can greatly impact the learning process. According to the results, we conclude that freezing more layers works only if the distributional similarity between the data from the new experience and data from the experience used to train the frozen layers is high or if the new task is simpler in terms of feature complexity. These results are aligned with the findings in Yosinski et al. (2014).

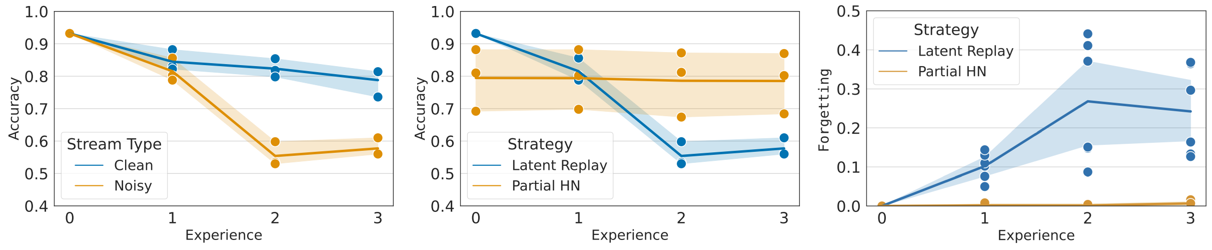


Figure 7: Left: test accuracy of the **first experience** over time in LR with different freezing depths and replay coefficient $c = 1$ for the noisy and clean versions of the same stream. Middle and Right: comparison of test accuracy and forgetting for the **first experience** between LR and Partial HN for different freezing depths in the noisy stream

.

4.2 Robustness to Noise

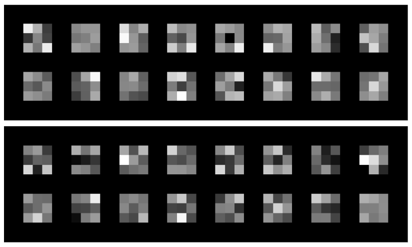


Figure 6: Visualization of the filters learned for experience 2 (above) and experience 3 (bottom) in a partial HN with freeze-depth=2 in a noisy stream.

The primary objective in this experiment is to evaluate the effectiveness of HNs in a setting where latent replay methods may struggle to retain performance. We first hypothesized that in the presence of significant distribution shifts between experiences, replaying older samples with newer ones may result in a severe degradation of performance in older experiences in LR methods. Since one of the key features of HNs is their input adaptability, we sought to investigate whether having separate convolution kernels per experience can help mitigate interference between experiences in such settings.

To construct the noisy streams, we randomly shuffle CIFAR-100 classes, select the first 20 classes, and build streams with five classes in each experience. Each experience applies a different set of transformations to the images in its dataset. These transformations are distinct and allow us to observe the impact of significant changes in the stream over time. As illustrated in Figure 5, in our experiments we apply the following transformations: Random Solarize to experience 2, Gaussian Blur to experience 3, and a combination of Contrast, Gaussian Blur, and GrayScale to experience 4.

After training different model versions on both noisy and normal streams, we noticed that the performance of LR with replay coefficient $c = 1$ is significantly harmed when the stream is noisy, especially after training on the third experience with the Gaussian Blurring noise. The issue remains despite using the same buffer size and achieving almost the same learning accuracy on the new experiences in both stream types (see Table 1). However, the test accuracy of the model in partial HNs remains constant in both "normal" and "noisy" streams (Figure 7). That said, the learning accuracy in partial HNs significantly varies to a large extent under the same training setting depending on the number of frozen layers. As presented in Table 1, all model versions in partial HNs versions have lower learning test accuracy compared to those of LR. In general, we experienced more difficulty in convergence in HN variations. Among the HN model version, ResNet-3 performs better on average compared to ResNet-2, 4 in terms of learning test accuracy. Moreover, we visualize the filters of a random convolutional layer in the third block of the model in partial HNs after the third experience in Figure 6. We can see that the model learns mostly different sets of filters for the new task, which can be the reason for the lower interference between experiences in partial HNs, while LR is forced to use a shared set of filters.

Model	**Stream**	$\mathbf{e_1}$	$\mathbf{e_2}$	$\mathbf{e_3}$	$\mathbf{e_4}$
ResNet-2	Normal	93.2 ± 0.2	94.3 ± 0.4	91.2 ± 0.4	92.2 ± 0.3
	Noisy	93.2 ± 0.2	91.2 ± 0.6	89.1 ± 1.3	91.2 ± 0.9
ResNet-2*	Normal	88.2 ± 1.4	77.2 ± 0.3	76.6 ± 0.7	64.6 ± 0.3
	Noisy	88.2 ± 1.4	76.6 ± 1.3	65.0 ± 2.1	50.0 ± 0.8
ResNet-3	Normal	93.1 ± 0.8	94.3 ± 0.1	89.9 ± 0.7	92.3 ± 0.4
	Noisy	93.1 ± 0.8	91.1 ± 0.9	88.0 ± 1.2	87.4 ± 0.7
ResNet-3*	Normal	81.0 ± 0.6	71.4 ± 0.5	70.3 ± 0.4	71.5 ± 0.4
	Noisy	81.0 ± 0.6	69.1 ± 1.1	66.2 ± 1.3	68.1 ± 0.9
ResNet-4	Normal	93.1 ± 0.4	91.3 ± 0.6	85.1 ± 0.2	89.7 ± 0.5
	Noisy	93.1 ± 0.4	86.4 ± 1.2	86.0 ± 0.6	80.1 ± 0.9
ResNet-4*	Normal	69.1 ± 0.3	68.6 ± 0.2	73.4 ± 0.2	72.1 ± 0.1
	Noisy	69.1 ± 0.3	68.4 ± 0.4	73.4 ± 0.6	67.5 ± 0.3

Table 1: Learning accuracy of each experience in LR and Partial HN in noisy and normal streams. Models in the partial HN strategy are marked with *. Results are obtained from three run with three random class ordering.

4.3 Split- CIFAR-100 and TinyImageNet

In this experiment, we test partial HNs and LR in two standard CL classification benchmarks. Both benchmarks are comprised of 20 experiences with 5 and 10 distinct classes in the CIFAR and TinyImagenet benchmarks, respectively. We use a buffer size of 200 for CIFAR-100 and 400 for Tiny-ImageNet. The complete comparison between partial HN and LR for ResNet-k, $0 \leq k \leq 4$ is demonstrated in Table 2. It is obvious from the results that the partial HN version of each ResNet-k outperforms its LR counterpart in terms of average test accuracy between experiences. More importantly, the running time in partial HNs for model versions with larger k's is much lower compared to a full hypernetwork. In both benchmarks, the HN-2 strategy which uses the ResNet-2 model achieves very similar average test accuracy to the full HN version. Therefore by freezing the first two blocks in the model we can achieve the same accuracy with approximately half the training time.

However, by looking at the forgetting metric, we realized that despite the low forgetting indicator, HNs achieve overall lower learning accuracy in the same training setting. This is affected by several factors such as the number of iterations, learning rate schedule, and other optimization settings. In Figure 8 we show the test accuracy for the first three experiments over time for freezing depth 3. The partial HN strategy is much more stable in terms of retraining accuracy compared to the LR strategies but has lower learning accuracy.

5 Related Work

Continual Learning methods are mainly categorized into five groups Wang et al. (2023): regularization, replay and optimization, representation and architecture. In regularization methods, the goal is to control catastrophic forgetting by adding a regularization term to the loss function that penalizes changes to the model parameters that would cause a large shift in the outputs of the previously learned tasks Kirkpatrick et al. (2017); Zenke et al. (2017); Li & Hoiem (2017). In rehearsal methods, the goal is to mitigate forgetting by storing a subset of the original data for each task and replaying it during training on subsequent tasks, either by interleaving it with new data or by training the model on the replayed data after training on the new data Rolnick et al. (2019); Chaudhry et al. (2019); Rebuffi et al. (2017);

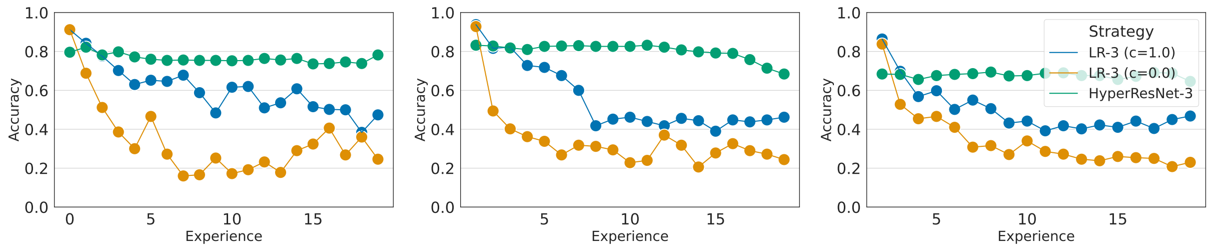


Figure 8: Test accuracy for the first (left), second(middle) and the third (right) experiences by freezing the first initial layers in ResNet-18 in the LR and partial HN methods. The coefficient c in the LR method indicates the exemplar replay coefficient.

Table 2: Average Classification Accuracy, Forgetting and Running Time measurements for the Split-CIFAR100 and Split-Imagenet Benchmarks. The number k in the LR-k and HN-k strategies indicates that the strategy uses ResNet-k model. "Naive" refers to a strategy that performs SGD updates on the model.

	CIFAR-100			**TinyImagenet**		
Strategy	ACA	Forgetting	Running Time	ACA	Forgetting	Running Time
Naive	33.7 ± 0.7	58.9 ± 2.1	$\sim$ 120 Min	20.5 ± 1.7	48.4 ± 0.9	$\sim$ 371 Min
EWC	29.7 ± 1.0	62.2 ± 0.9	$\sim$ 230 Min	22.4 ± 1.1	34.5 ± 0.5	$\sim$ 431 Min
LR-1	48.1 ± 1.2	43.0 ± 0.8	$\sim$ 165 Min	30.6 ± 0.9	38.1 ± 2.1	$\sim$ 323 Min
HN-1	$\mathbf{61.9 \pm 1.7}$	8.1 ± 0.6	$\sim$ 508 Min	$\mathbf{35.2 \pm 0.3}$	08.0 ± 1.4	$\sim$ 905 Min
LR-2	51.1 ± 0.8	40.3 ± 1.1	$\sim$ 134 Min	30.0 ± 0.7	38.8 ± 1.2	$\sim$ 245 Min
HN-2	$\mathbf{62.8 \pm 1.6}$	4.1 ± 0.5	$\sim$ 388 Min	$\mathbf{39.9 \pm 0.4}$	10.2 ± 2.2	$\sim$ 660 Min
LR-3	50.1 ± 2.2	42.4 ± 1.3	$\sim$ 111 Min	26.5 ± 0.6	36.9 ± 0.9	$\sim$ 204 Min
HN-3	$\mathbf{58.8 \pm 1.0}$	7.4 ± 0.9	$\sim$ 243 Min	$\mathbf{31.4 \pm 1.8}$	5.2 ± 0.7	$\sim$ 416 Min
LR-4	51.4 ± 1.3	27.3 ± 2.4	$\sim$ 89 Min	31.8 ± 1.1	25.9 ± 2.0	$\sim$ 143 Min
HN-4	$\mathbf{73.6 \pm 0.9}$	3.1 ± 0.4	$\sim$ 114 Min	$\mathbf{36.1 \pm 1.1}$	10.0 ± 1.1	$\sim$ 176 Min
HN-F	64.2 ± 1.2	5.5 ± 1.4	$\sim$ 636 Min	39.9 ± 1.7	7.2 ± 1.8	$\sim$ 1200 Min

Pellegrini et al. (2020). Optimization-based methods attempt to alleviate forgetting by manipulating the optimization process either via projecting gradients or customized optimizer Lopez-Paz & Ranzato (2017); Farajtabar et al. (2020). In representation-based methods, the main ideas is to learn representations that can boost continual learning for example by learning more sparse representations as in Javed & White (2019) or by learning richer representations for transfer learning via pre-training and self-supervised learning Gallardo et al. (2021); Mehta et al. (2021). In the architecture-based methods, the focus is on the role of the model's architecture in CL with the assumption that model needs to learn task-specific parameters which is itself split to sub-groups: parameter isolation and dynamic architecture. In parameter isolation, the model architecture is fixed and different subsets of parameters are assigned to different tasks, for example by masking parameters Serra et al. (2018); Mallya et al. (2018) or model decomposition such as Miao et al. (2022); Yoon et al. (2019); Kanakis et al. (2020). In dynamic architecture methods, new modules are added over time for solving newer tasks such as Rusu et al. (2016); Aljundi et al. (2017); Xu & Zhu (2018). Our method is can be categorized into the parameter isolation sub-group in the architecture group.

Hypernetworks are a type of neural network architecture that have gained significant popularity in recent years. The weight generation mechanism in HNs enables them to be applied to several applications such as neural architecture search Zhang et al. (2018) and conditional neural networks Galanti & Wolf (2020). Although the origin of hypernetworks dates back to the work of Hinton and Salakhutdinov in the 1990s Schmidhuber (1992), it was not until recently that they became a popular topic of research in deep learning Ha et al. (2016). In the context of CL, the work from Von Oswald et al. (2019) proposed to use hypernetwork to generate the full weights of a neural network. Morever, Ehret et al. (2020) proposed to use hypernetwork for more effective continual learning compared to the weight regularization-based methods. A probabilistic version of hypernetworks for CL for task-conditioned CL was proposed by Henning et al. (2021).

Latent Replay methods propose to store or generate activations from intermediate layers of a neural network, which are then replayed at the same intermediate layers to prevent forgetting when new tasks are learned during the CL process. Intuitively, Latent Replay methods assume that low-level representations can be better shared across tasks and, hence, require less adaptation. To this end, low layers are either regularized Liu et al. (2020), trained at a slower pace Pellegrini et al. (2020), or frozen, as in Van de Ven et al. (2020); Hayes et al. (2019). Freezing the low layers significantly reduces the computational cost of CL by reducing the number of trainable parameters. These methods rely on the assumption that feature extractors can produce meaningful features and should be preferred in applications with limited compute. In Ostapenko et al. (2022), the efficacy of pre-trained vision models for downstream continual learning is studied, exploring the trade-off between CL in raw data and latent spaces, comparing pre-trained models in benchmarking scenarios, and identifying research directions to increase efficacy.

Transfer Learning is a highly popular paradigm in deep learning that enables sample-efficient learning by partially or completely fine-tuning a pre-trained model for a novel task. Many researchers have studied transfer learning to understand *when and how* transfer learning works. Yosinski et al. Yosinski et al. (2014) quantify the generality vs. specificity of deep convolutional neural networks, showing transferability issues and boosting generalization by transferring features. They demonstrate that the first layer of AlexNet can be frozen when transferring between natural and man-made subsets of ImageNet without performance impairment, but freezing later layers produces a substantial drop in accuracy. Neyshabur et al. Neyshabur et al. (2020) analyze the transfer of knowledge in deep learning and show that successful transfer results from learning low-level statistics of data. They also show that models trained from pre-trained weights stay in the same basin in the loss landscape. Kornblith et al. Kornblith et al. (2019a) examine methods for comparing neural network representations, show the limitations of canonical correlation analysis (CCA), and introduce a similarity index based on centered kernel alignment (CKA) that can identify correspondences between representations in networks trained from different initializations. The work by Kornblith et al. Kornblith et al. (2019b) evaluates the relationship between architecture and transfer learning by comparing various networks in different classification benchmarks. They find that pre-training on ImageNet provides minimal benefits for fine-grained image classification tasks. Moreover, He et al. He et al. (2019) show that transfer even between similar tasks does not necessarily improve performance, and random initialization can achieve a better solution given enough iterations.

6 Conclusion and Discussion

This study explores the application of hypernetworks for partial weight generation in the context of continual learning. The research question that guided our investigation was *whether it is necessary to generate all the weights of a neural network when using hypernetworks* to learn from a stream of experiences. Our approach to answering this question was inspired by recent work in latent replay for CL, and we hypothesized that generating only a subset of the model parameters via hypernetworks could be a practical method for preserving previous knowledge while adapting to new information.

To test this hypothesis, we proposed decomposing the model into two parts and generating the weights of the second part of the main neural network. By generating only a subset of the model parameters in the final layers of the neural network, we showed that we could speed up the learning process and thus reduce the computational costs, and preserve previous knowledge, compared to the full-weight generation approach. Furthermore, we studied the impact of different decomposition depths in various settings on the final performance of both latent replay and hypernetwork variants. To further justify the use of partial hypernetworks, we also showed through our experiments that hypernetworks are more robust to strong noise in the stream in terms of negative interference between tasks compared to latent replay. The low interference in hypernetworks suggests that partially generating weights can effectively preserve previous knowledge when data from noisy distribution are replayed with clean data. Our findings indicate that partial weight generation via hypernetworks is a promising approach for preserving previous knowledge while adapting to new information.

In spite of their ability to preserve knowledge effectively, our findings indicate that partial HNs have lower learning accuracy when trained under the same training conditions. Additionally, our research only focused on using hypernetworks for weight generation in the second part of the model. A promising avenue for future inquiry would be to explore methods for enhancing learning accuracy while reducing training time in HN-based CL strategies. Alternatively, we could investigate the feasibility of using adaptive layers between pre-trained models to generate model weights partially.

In summary, this research contributes to our understanding of the effectiveness of partial hypernetworks in continual learning and provides practical guidelines for optimizing them. We believe that our work will facilitate the development of more computationally efficient algorithms for continual learning using hypernetworks. We encourage further investigation into the relationship between hypernetworks and continual learning, and we hope that our research will inspire future work in this area.

REFERENCES

Rahaf Aljundi, Punarjay Chakravarty, and Tinne Tuytelaars. Expert gate: Lifelong learning with a network of experts. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 3366–3375, 2017.

Jacob Beck, Matthew Thomas Jackson, Risto Vuorio, and Shimon Whiteson. Hypernetworks in meta-reinforcement learning. *arXiv preprint arXiv:2210.11348*, 2022.

Magdalena Biesialska, Katarzyna Biesialska, and Marta R Costa-Jussa. Continual lifelong learning in natural language processing: A survey. *arXiv preprint arXiv:2012.09823*, 2020.

Andrew Brock, Theodore Lim, James M Ritchie, and Nick Weston. Smash: one-shot model architecture search through hypernetworks. *arXiv preprint arXiv:1708.05344*, 2017.

Arslan Chaudhry, Marcus Rohrbach, Mohamed Elhoseiny, Thalaiyasingam Ajanthan, Puneet K Dokania, Philip HS Torr, and Marc'Aurelio Ranzato. On tiny episodic memories in continual learning. *arXiv preprint arXiv:1902.10486*, 2019.

François Cubélier et al. Master thesis: Sparse hypernetworks for multitasking. 2022.

Matthias De Lange, Rahaf Aljundi, Marc Masana, Sarah Parisot, Xu Jia, Aleš Leonardis, Gregory Slabaugh, and Tinne Tuytelaars. A continual learning survey: Defying forgetting in classification tasks. *IEEE transactions on pattern analysis and machine intelligence*, 44(7):3366–3385, 2021.

Giorgos Demosthenous and Vassilis Vassiliades. Continual learning on the edge with tensorflow lite. *arXiv preprint arXiv:2105.01946*, 2021.

Benjamin Ehret, Christian Henning, Maria R Cervera, Alexander Meulemans, Johannes Von Oswald, and Benjamin F Grewe. Continual learning in recurrent neural networks. *arXiv preprint arXiv:2006.12109*, 2020.

Mehrdad Farajtabar, Navid Azizan, Alex Mott, and Ang Li. Orthogonal gradient descent for continual learning. In *International Conference on Artificial Intelligence and Statistics*, pp. 3762–3773. PMLR, 2020.

Robert M French. Catastrophic forgetting in connectionist networks. *Trends in cognitive sciences*, 3(4):128–135, 1999.

Tomer Galanti and Lior Wolf. On the modularity of hypernetworks. *Advances in Neural Information Processing Systems*, 33:10409–10419, 2020.

Jhair Gallardo, Tyler L Hayes, and Christopher Kanan. Self-supervised training enhances online continual learning. *arXiv preprint arXiv:2103.14010*, 2021.

Gabriele Graffieti, Davide Maltoni, Lorenzo Pellegrini, and Vincenzo Lomonaco. Generative negative replay for continual learning. *arXiv preprint arXiv:2204.05842*, 2022.

Gunshi Gupta, Karmesh Yadav, and Liam Paull. Look-ahead meta learning for continual learning. *Advances in Neural Information Processing Systems*, 33:11588–11598, 2020.

David Ha, Andrew Dai, and Quoc V Le. Hypernetworks. *arXiv preprint arXiv:1609.09106*, 2016.

Trevor Hastie, Robert Tibshirani, Jerome H Friedman, and Jerome H Friedman. *The elements of statistical learning: data mining, inference, and prediction*, volume 2. Springer, 2009.

Tyler L Hayes, Kushal Kafle, Robik Shrestha, Manoj Acharya, and Christopher Kanan. Remind your neural network to prevent catastrophic forgetting (october) arxiv e-prints, page. *arXiv preprint arXiv:1910.02509*, 2019.

Kaiming He, Ross Girshick, and Piotr Dollár. Rethinking imagenet pre-training. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pp. 4918–4927, 2019.

Christian Henning, Maria Cervera, Francesco D'Angelo, Johannes Von Oswald, Regina Traber, Benjamin Ehret, Seijin Kobayashi, Benjamin F Grewe, and João Sacramento. Posterior meta-replay for continual learning. *Advances in Neural Information Processing Systems*, 34:14135–14149, 2021.

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

Menelaos Kanakis, David Bruggemann, Suman Saha, Stamatios Georgoulis, Anton Obukhov, and Luc Van Gool. Reparameterizing convolutions for incremental multi-task learning without task interference. In *Computer Vision–ECCV 2020: 16th European Conference, Glasgow, UK, August 23–28, 2020, Proceedings, Part XX 16*, pp. 689–707. Springer, 2020.

K Khetarpal, M Riemer, I Rish, and D Precup. Towards continual reinforcement learning: A review and perspectives. arxiv. *arXiv preprint arXiv:2012.13490*, 2020.

Gyuhak Kim, Changnan Xiao, Tatsuya Konishi, Zixuan Ke, and Bing Liu. A theoretical study on solving continual learning. *arXiv preprint arXiv:2211.02633*, 2022.

James Kirkpatrick, Razvan Pascanu, Neil Rabinowitz, Joel Veness, Guillaume Desjardins, Andrei A Rusu, Kieran Milan, John Quan, Tiago Ramalho, Agnieszka Grabska-Barwinska, et al. Overcoming catastrophic forgetting in neural networks. *Proceedings of the national academy of sciences*, 114(13):3521–3526, 2017.

Simon Kornblith, Mohammad Norouzi, Honglak Lee, and Geoffrey Hinton. Similarity of neural network representations revisited. In *International Conference on Machine Learning*, pp. 3519–3529. PMLR, 2019a.

Simon Kornblith, Jonathon Shlens, and Quoc V Le. Do better imagenet models transfer better? In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pp. 2661–2671, 2019b.

Alex Krizhevsky, Geoffrey Hinton, et al. Learning multiple layers of features from tiny images. 2009.

Timothée Lesort, Vincenzo Lomonaco, Andrei Stoian, Davide Maltoni, David Filliat, and Natalia Díaz-Rodríguez. Continual learning for robotics: Definition, framework, learning strategies, opportunities and challenges. *Information fusion*, 58:52–68, 2020.

Zhizhong Li and Derek Hoiem. Learning without forgetting. *IEEE transactions on pattern analysis and machine intelligence*, 40(12):2935–2947, 2017.

Xialei Liu, Chenshen Wu, Mikel Menta, Luis Herranz, Bogdan Raducanu, Andrew D Bagdanov, Shangling Jui, and Joost van de Weijer. Generative feature replay for class-incremental learning. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops*, pp. 226–227, 2020.

Vincenzo Lomonaco, Lorenzo Pellegrini, Andrea Cossu, Antonio Carta, Gabriele Graffieti, Tyler L Hayes, Matthias De Lange, Marc Masana, Jary Pomponi, Gido M Van de Ven, et al. Avalanche: an end-to-end library for continual learning. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 3600–3610, 2021.

David Lopez-Paz and Marc'Aurelio Ranzato. Gradient episodic memory for continual learning. *Advances in neural information processing systems*, 30, 2017.

Arun Mallya, Dillon Davis, and Svetlana Lazebnik. Piggyback: Adapting a single network to multiple tasks by learning to mask weights. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pp. 67–82, 2018.

Sanket Vaibhav Mehta, Darshan Patil, Sarath Chandar, and Emma Strubell. An empirical investigation of the role of pre-training in lifelong learning. *arXiv preprint arXiv:2112.09153*, 2021.

Zichen Miao, Ze Wang, Wei Chen, and Qiang Qiu. Continual learning with filter atom swapping. In *International Conference on Learning Representations*, 2022.

Kevin P Murphy. *Probabilistic machine learning: an introduction*. MIT press, 2022.

Yuval Netzer, Tao Wang, Adam Coates, Alessandro Bissacco, Bo Wu, and Andrew Y Ng. Reading digits in natural images with unsupervised feature learning. 2011.

Behnam Neyshabur, Hanie Sedghi, and Chiyuan Zhang. What is being transferred in transfer learning? *Advances in neural information processing systems*, 33:512–523, 2020.

Phuoc Nguyen, Truyen Tran, Ky Le, Sunil Gupta, Santu Rana, Dang Nguyen, Trong Nguyen, Shannon Ryan, and Svetha Venkatesh. Fast conditional network compression using bayesian hypernetworks. In *Machine Learning and Knowledge Discovery in Databases. Research Track: European Conference, ECML PKDD 2021, Bilbao, Spain, September 13–17, 2021, Proceedings, Part III 21*, pp. 330–345. Springer, 2021.

Oleksiy Ostapenko, Timothee Lesort, Pau Rodríguez, Md Rifat Arefin, Arthur Douillard, Irina Rish, and Laurent Charlin. Foundational models for continual learning: An empirical study of latent replay. *arXiv preprint arXiv:2205.00329*, 2022.

German I Parisi, Ronald Kemker, Jose L Part, Christopher Kanan, and Stefan Wermter. Continual lifelong learning with neural networks: A review. *Neural networks*, 113:54–71, 2019.

Lorenzo Pellegrini, Gabriele Graffieti, Vincenzo Lomonaco, and Davide Maltoni. Latent replay for real-time continual learning. In *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 10203–10209. IEEE, 2020.

Vinay V Ramasesh, Ethan Dyer, and Maithra Raghu. Anatomy of catastrophic forgetting: Hidden representations and task semantics. *arXiv preprint arXiv:2007.07400*, 2020.

Sylvestre-Alvise Rebuffi, Alexander Kolesnikov, Georg Sperl, and Christoph H Lampert. icarl: Incremental classifier and representation learning. In *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*, pp. 2001–2010, 2017.

David Rolnick, Arun Ahuja, Jonathan Schwarz, Timothy Lillicrap, and Gregory Wayne. Experience replay for continual learning. *Advances in Neural Information Processing Systems*, 32, 2019.

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.

Jürgen Schmidhuber. Learning to control fast-weight memories: An alternative to dynamic recurrent networks. *Neural Computation*, 4(1):131–139, 1992.

Joan Serra, Didac Suris, Marius Miron, and Alexandros Karatzoglou. Overcoming catastrophic forgetting with hard attention to the task. In *International Conference on Machine Learning*, pp. 4548–4557. PMLR, 2018.

Hanul Shin, Jung Kwon Lee, Jaehong Kim, and Jiwon Kim. Continual learning with deep generative replay. *Advances in neural information processing systems*, 30, 2017.

Sebastian Thrun and Tom M Mitchell. Lifelong robot learning. *Robotics and autonomous systems*, 15(1-2):25–46, 1995.

Ahmet Üstün, Arianna Bisazza, Gosse Bouma, Gertjan van Noord, and Sebastian Ruder. Hyper-x: A unified hypernetwork for multi-task multilingual transfer. *arXiv preprint arXiv:2205.12148*, 2022.

Gido M Van de Ven and Andreas S Tolias. Three scenarios for continual learning. *arXiv preprint arXiv:1904.07734*, 2019.

Gido M Van de Ven, Hava T Siegelmann, and Andreas S Tolias. Brain-inspired replay for continual learning with artificial neural networks. *Nature communications*, 11(1):4069, 2020.

Johannes Von Oswald, Christian Henning, João Sacramento, and Benjamin F Grewe. Continual learning with hypernetworks. *arXiv preprint arXiv:1906.00695*, 2019.

Liyuan Wang, Mingtian Zhang, Zhongfan Jia, Qian Li, Chenglong Bao, Kaisheng Ma, Jun Zhu, and Yi Zhong. Afec: Active forgetting of negative transfer in continual learning. *Advances in Neural Information Processing Systems*, 34:22379–22391, 2021.

Liyuan Wang, Xingxing Zhang, Hang Su, and Jun Zhu. A comprehensive survey of continual learning: Theory, method and application. *arXiv preprint arXiv:2302.00487*, 2023.

Han Xiao, Kashif Rasul, and Roland Vollgraf. Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms. *arXiv preprint arXiv:1708.07747*, 2017.

Ju Xu and Zhanxing Zhu. Reinforced continual learning. *Advances in Neural Information Processing Systems*, 31, 2018.

Jaehong Yoon, Saehoon Kim, Eunho Yang, and Sung Ju Hwang. Scalable and order-robust continual learning with additive parameter decomposition. *arXiv preprint arXiv:1902.09432*, 2019.

Jason Yosinski, Jeff Clune, Yoshua Bengio, and Hod Lipson. How transferable are features in deep neural networks? *Advances in neural information processing systems*, 27, 2014.

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

Chris Zhang, Mengye Ren, and Raquel Urtasun. Graph hypernetworks for neural architecture search. *arXiv preprint arXiv:1810.05749*, 2018.

Michael Zhang, James Lucas, Jimmy Ba, and Geoffrey E Hinton. Lookahead optimizer: k steps forward, 1 step back. *Advances in neural information processing systems*, 32, 2019.

A Model Details

In this section, we elaborate on the architecture details of both the hypernetwork and the main model.

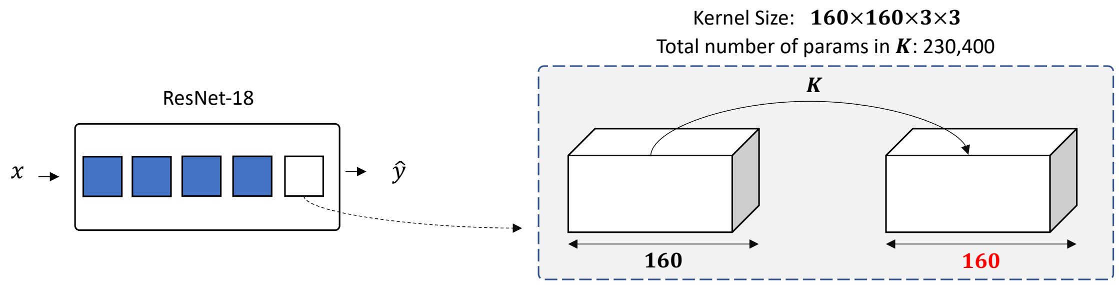


Figure 9: Number of parameters in a convolution module in the final residual layer of the main model.

A.1 Architecture

We employ a slimmed version of ResNet-18 as our main model, which comprises a single convolutional layer, four residual blocks, and a linear classifier. The model consists mainly of convolution, batch-norm, and linear module types. For the LR strategy, we utilized a multi-head classifier in the final layer, which we implemented using the `MultiHeadClassifier` module from Avalanche [1]. In the HN versions, the weights of the linear classifier are generated for each experience separately and we set the number of output neurons equal to the maximum number of classes in each experience.

For the hypernetwork, we use an MLP with two hidden layers. The input to the hypernetwork is the task ID. The naive way to generate weights of a model would be to generate all weights with a linear mapping directly after the second hidden layer. However, this method can become very expensive if the total number of parameters in the main model is high. For example, if the main model has $300,000$ parameters, and the second hidden layer of the hypernetwork has 100 units, the number of parameters in the final linear layer would be $300,000 \times 100 = 30,000,000$ parameters. Instead, like most previous methods Von Oswald et al. (2019); Henning et al. (2021) we propose a different way to generate the weights of the main model according to the structure of the architecture. In our setting, the final layer of the MLP outputs the weights of the main model for each module separately, thus, the total number of heads in the HN is equal to the total number of modules in the main model. We use three types of HN heads convolutional, batch-norm, and linear. Since most parameters are inside the final convolutional modules, we propose a simple yet efficient channel-wise weight generation for the convolutional layers that can be used for any convolutional neural network.

A.2 Channel-wise Weight Generation

The number of parameters for a convolution module with a input channels, b output channels, and a kernel size of k equals $a \times b \times k \times k$. If the number of units in the second hidden layer is h, then by direct weight generation, we will have $h \times a \times b \times k \times k$ parameters. Making h smaller may work up to a certain level, but if h is too small the HN will not be able to generate good filters. Therefore, we need to solve this issue by handling a or b. We use a look-up table for a to reduce its dimensionality. By assigning a learnable look-up vector with dimension $d < a$, we can condition the weight generation for the kernel of each output channel with $(h + d) \times b \times k \times k$. We show this process in Figure 10. Usually, very small d's (e.g. $d = 8$) work well. Hence the final number of parameters in the HN head will be $d \times a$ for the learnable lookup table plus $(h + d) \times b \times k \times k$ which is in most cases much smaller than the original number of parameters. Moreover, this type of weight generation also considers the structure of convolutional layers. We compare levels of compression in the final by setting different values to d in Table 3.

B Ablation Study

In this section, we conduct an ablation study for the look-ahead mechanism and the computation of the second-order derivatives.

[1]`https://github.com/ContinualAI/avalanche/blob/master/avalanche/models/dynamic_modules.py`

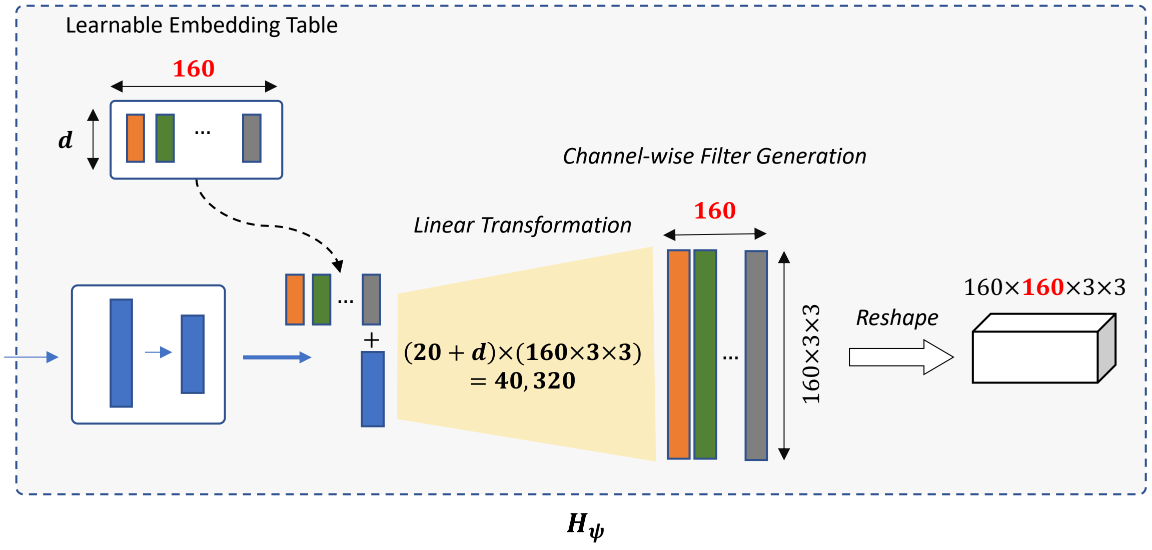

Learnable Embedding Table
160
d
Channel-wise Filter Generation
Linear Transformation
160
t
(20 + d)×(160×3×3) = 40,320
160×3×3
Reshape
160×160×3×3
H_ψ

Figure 10: Channel-wise weight generation for a convolutional layer with 160 input and output channels and a kernel size of 3×3.

Table 3: Compression level for different d's. Number of hidden units in the second hidden layer is equal to 32.

d	Number of Parameters	Compression Level
4	451661	59%
8	496489	55%
16	586145	47%
32	765457	31%
64	1124081	0%

B.1 Look-Ahead vs. No Look-Ahead

The look-ahead mechanism enables a more accurate calculation of the gradients for the hypernetwork. Here, we show that applying look-ahead-based updates can be beneficial, especially for ResNet-4, without significantly increasing computation time. In Table 4, we compare partial HNs with and without look-ahead updates for different versions of ResNet-k for the same random seed and class order in the stream. The results show that in most cases, the look-ahead update improves the ACA at the end of the stream. Furthermore, we visualize the test accuracy for experiences 1 to 8 in a random run for both update modes in Figure 11. The visualizations indicate that except for the second, in the rest of the experiences, the strategy with look-ahead update performs overall better.

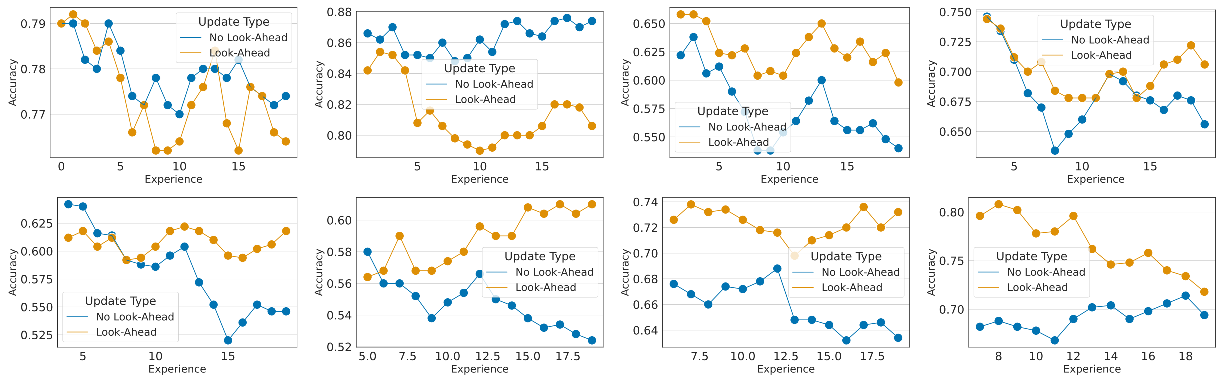


Figure 11: Test accuracy for experience 1 (top-left) to experience 8 (bottom-right) with and without look-ahead updates. The class order and model-initialization are the same for both modes.

Model	Update Type	ACA	Learning Accuracy	Running Time
ResNet-0	No Look-Ahead	62.7	65.0	407 Min
	Look-Ahead	62.8	66.5	636 Min
ResNet-1	No Look-Ahead	63.3	68.0	377 Min
	Look-Ahead	61.5	66.2.0	509 Min
ResNet-2	No Look-Ahead	61.0	64.2	265 Min
	Look-Ahead	63.2	66.8	389 Min
ResNet-3	No Look-Ahead	57.0	59.3	165 Min
	Look-Ahead	59.4	64.4	243 Min
ResNet-4	No Look-Ahead	66.9	73.4	87 Min
	Look-Ahead	74.3	75.6	115 Min

Table 4: Look-Ahead vs No Look-Ahead

B.2 First Order vs. Second-Order

Since the computation of the second-order derivatives can be computationally very expensive, we compare both first-order and second-order based estimations of the gradients for the hypernetwork. As demonstrated in Table 5, for some model versions, the second-order estimation can be slightly better, while in others, the first-order can even perform better in terms of ACA. However, the running time for the second-order is significantly longer. Therefore, considering the huge gap in the running time, a first-order estimation is a more reasonable choice.

Model	Update Type	ACA	Learning Accuracy	Running Time
ResNet-0	First-Order	63.8	66.5	636 Min
	Second-Order	64.1	66.2	2406 Min
ResNet-1	First-Order	66.5	66.2	509 Min
	Second-Order	61.9	66.1	1626 Min
ResNet-2	First-Order	63.2	66.8	389 Min
	Second-Order	64.0	64.7	1104 Min
ResNet-3	First-Order	59.4	64.6	243 Min
	Second-Order	56.6	61.9	585 Min
ResNet-4	First-Order	74.3	75.6	115 Min
	Second-Order	73.1	75.0	122 Min

Table 5: First Order vs. Second Order

C Memory Usage

In this section we compare the memory usage between different versions of latent replay and partial hypernetworks in Table 6. For LR, the memory usage is calculated for the exemplars stored in the replay buffer, and for HN, the memory usage is equal to the size of the hypernetwork since we need to store the converged hypernetwok from the previous experience for computing the regularization term in the new experience. We assume 32-bit tensors in our calculations.

The memory usage for exemplars in LR is affected by both increasing the size of the input and changing the architecture, in particular by changing the number of channels in a convolutional neural network. In HN, the memory usage is determined by the size of the main network, which can be reduced by partially generating the weights of the main model.

Table 6: Memory usage comparison between different version of LR and HN. The buffer size for the CIFAR-100 and TinyImageNet datasets are 200 and 400 respectively.

	Latent Replay			**Partial Hypernetwork**		
	Model	Exemplar Shape	Memory Usage	Model	# of HN Parameters	Memory Usage
CIFAR-100	LR-0	$3 \times 32 \times 32$	2.34 MB	HN-F	$1,272,877$	4.86 MB
	LR-1	$20 \times 32 \times 32$	15.62 MB	HN-1	$1,217,122$	4.64 MB
	LR-2	$4 \times 16 \times 16$	7.81 MB	HN-2	$1,119,522$	4.27 MB
	LR-3	$80 \times 8 \times 8$	3.91 MB	HN-3	$924,322$	3.53 MB
	LR-4	$160 \times 4 \times 4$	1.95 MB	HN-4	$533,922$	2.04 MB
	Model	Exemplar Shape	Memory Usage	Model	# of HN Parameters	Memory Usage
TinyImageNet	LR-0	$3 \times 64 \times 64$	18.75 MB	HN-F	$1,272,877$	4.86 MB
	LR-1	$20 \times 64 \times 64$	125.0 MB	HN-1	$1,217,122$	4.64 MB
	LR-2	$40 \times 32 \times 32$	62.50 MB	HN-2	$1,119,522$	4.27 MB
	LR-3	$80 \times 16 \times 16$	31.25 MB	HN-3	$924,322$	3.53 MB
	LR-4	$160 \times 8 \times 8$	15.62 MB	HN-4	$533,922$	2.04 MB

D ALGORITHM

In Algorithm 1 we present the steps for training a partial hypernetwork over a CL stream.

Algorithm 1 Proposed algorithm for training a partial hypernetwork with look-ahead updates.

```
1: Input: CL stream $\mathcal{S}$, learning rates $\alpha$ and $\beta$, regularization factor $\lambda$
2: Initialize parameters of the main model's stateful layers $\phi$
3: Initialize hypernetwork parameters $\omega$
4: for each experience $e_i$ in the CL stream $\mathcal{S}$ do
5:     if $i = 1$ then                                        ▷ Train with SGD updates
6:         while not converged do
7:             Sample a mini-batch $b$ from $e_i$
8:             Compute gradients for $\phi$: $g_\phi \leftarrow \nabla_\phi \mathcal{L}(b, \{\phi, \omega\})$
9:             Compute gradients for $\omega$: $g_\omega \leftarrow \nabla_\omega \mathcal{L}(b, \{\phi, \omega\})$
10:            Update the main model's stateful parameters $\phi \leftarrow \phi - \alpha g_\phi$
11:            Update hypernetwork parameters $\omega \leftarrow \omega - \beta g_\omega$
12:        end while
13:        Freeze parameters $\phi$
14:    else                                                   ▷ Update with look-ahead mechanism
15:        while not converged do
16:            Sample a mini-batch $b$ from $e_i$
17:            Compute gradients $g_\omega^{(1)} \leftarrow \nabla_\omega \mathcal{L}(b, \omega)$
18:            Make virtual update on $\omega$: $\omega' \leftarrow \omega - \beta g_\omega$
19:            Compute gradients using virtual parameters $\omega'$: $g_\omega^{(2)} \leftarrow \nabla_\omega \mathcal{L}(\mathcal{B}, \omega')$
20:            Compute the average of $g_\omega^{(1)}$ and $g_\omega^{(2)}$: $\overline{g}_\omega = \frac{g_\omega^{(1)} + g_\omega^{(2)}}{2}$
21:            Update hypernetwork parameters $\omega \leftarrow \omega - \beta \overline{g}_\omega$
22:        end while
23:    end if
24: end for
```

E GRADIENT SIMILARITY

In Figure 12 we demonstrate the cosine similarity of the hypernetwork gradients $g_\omega^{(1)}$ (from the cross-entropy loss) and $g_\omega^{(2)}$ (from the virtual update) in HN-4, as computed in Algorithm 1 in the first epoch of the second experience. The plots show negative similarity regardless of the class order or the model initialization which are affected by the random seeds.

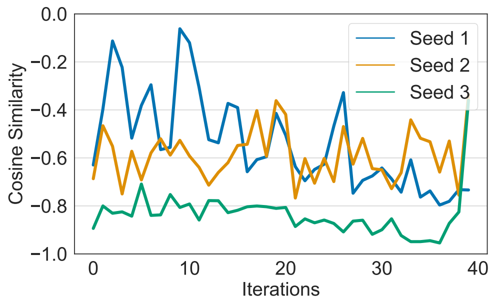


Figure 12: Cosine similarity between gradients $g_\omega^{(1)}$ and $g_\omega^{(2)}$.

F HYPERPARAMETER SELECTION

We carried out a hyper-parameter selection at the stream level. For common hyper-parameters like learning rate, we implemented a grid search and assessed the model's final performance on the test set of each stream across three runs (using 3 random seeds). In our experiments, we used a learning rate of 0.001 for both models.

For the hypernetwork, we executed a grid-search, opting for the minimum embedding size capable of attaining a similar performance. This approach allowed us to minimize the hypernetwork's number of parameters. In our experiments, we used dimension sizes 32, 50, 32 and 32 for the embedding dimension the three consecutive layers of the multi-layer peceptron respectively. We set the hypernetwork's regularization coefficient to 0.5.