Adaptive Plasticity Improvement for Continual Learning

Yan-Shuo Liang and Wu-Jun Li*
National Key Laboratory for Novel Software Technology,
Department of Computer Science and Technology, Nanjing University, P. R. China
liangys@smail.nju.edu.cn, liwujun@nju.edu.cn

Abstract

Many works have tried to solve the catastrophic forgetting (CF) problem in continual learning (lifelong learning). However, pursuing non-forgetting on old tasks may damage the model's plasticity for new tasks. Although some methods have been proposed to achieve stability-plasticity trade-off, no methods have considered evaluating a model's plasticity and improving plasticity adaptively for a new task. In this work, we propose a new method, called adaptive plasticity improvement (API), for continual learning. Besides the ability to overcome CF on old tasks, API also tries to evaluate the model's plasticity and then adaptively improve the model's plasticity for learning a new task if necessary. Experiments on several real datasets show that API can outperform other state-of-the-art baselines in terms of both accuracy and memory usage.

1. Introduction

Continual learning is a challenging setting in which agents learn multiple tasks sequentially [21]. However, neural network models lack the ability to perform continual learning. Specifically, many studies [15, 21] have shown that directly training a network on a new task makes the model forget the old knowledge. This phenomenon is often called catastrophic forgetting (CF) [10, 21].

Continual learning models need to overcome CF, which is referred to as stability [21]. Many types of works are proposed for stability. For example, regularization-based methods [13, 35] add a penalty to the loss function and minimize penalty loss with new task loss together for overcoming CF. Memory-based methods [5, 6, 24, 29] maintain a memory to save the information of the old tasks and use saved information to keep old task performance. Expansion-based methods [12, 16] expand the network's architecture and usually freeze old tasks' parameters to overcome CF.

However, having stability alone fails to give the model continual learning ability. The model also needs plasticity to learn new tasks in continual learning. The term plasticity came from neuroscience and was originally used to describe the brain's ability to yield physical changes in the neural structure. Plasticity allows us to learn, remember, and adapt to dynamic environments [22]. In neural networks, plasticity is used to describe the ability of a network to change itself for learning new tasks. However, existing works [17, 18, 30] show that when overcoming CF for stability, the model's plasticity will decrease, which will affect the performance of the model for learning new tasks. Specifically, regularization-based methods and memory-based methods use penalty or memory to constrain the parameters when the model learns a new task. When the number of old tasks increases, the constraints for the model parameters should become stronger and stronger to ensure stability. As a result, the model's plasticity for learning new tasks decreases. Expansion-based methods [28, 32] increase the model's plasticity by expanding additional parameters. However, most of these methods freeze the old part of the network, making the old part of the network underutilized. Furthermore, all these methods do not consider how to evaluate the model's plasticity and improve it adaptively.

When overcoming CF, the model should improve its plasticity if it finds that current plasticity is insufficient to learn the new task. In this work, we propose a new method, called adaptive plasticity improvement (API), for continual learning. The main contributions of API are as follows:

- API overcomes CF through a new memory-based method called dual gradient projection memory (DualGPM), which learns a gradient subspace that can represent the gradients of old tasks.
- Based on DualGPM, API evaluates the model's plasticity for a new task by average gradient retention ratio (AGRR) and improves the model's plasticity adaptively for a new task if necessary.
- Experiments on several real datasets show that API can outperform other state-of-the-art baselines in terms of accuracy and memory usage.

*Wu-Jun Li is the corresponding author.

2. Problem Formulation and Related Work

2.1. Problem Formulation

We consider the supervised continual learning setting where T tasks are presented to the model sequentially. Each task has a dataset $\mathcal{D}_t = \{(\boldsymbol{x}_i^t, y_i^t)\}_{i=1}^{N_t}$ sampled from a latent distribution $\mathfrak{D}_t$, where $\boldsymbol{x}_i^t$ represents the input data point and y_i^t represents its class label. A neural network model $f(\cdot, \boldsymbol{\Theta})$ with parameters $\boldsymbol{\Theta}$ is trained on these tasks sequentially. The aim is to minimize the average loss of all tasks, that is

$$\mathcal{L} = \frac{1}{T}\sum_{t=1}^{T}\mathbb{E}_{(\boldsymbol{x}_i^t, y_i^t)\sim\mathfrak{D}_t}\left[l(f(\boldsymbol{x}_i^t;\boldsymbol{\Theta}), y_i^t)\right]. \quad (1)$$

Here, $l(\cdot,\cdot)$ is the loss function (e.g., cross-entropy). When learning a new task t, the model has no access to the data of the previous $t-1$ tasks and it needs to learn new tasks while maintaining the performance of old tasks. Like many recent works [14, 17], we assume the task identity is available in both training and inference stages.

2.2. Related Work

Existing continual learning methods can be divided into three main categories: regularization-based methods, memory-based methods and expansion-based methods.

Regularization-based Methods These methods use a penalty loss (regularization) to prevent important parameters of old tasks from changing too much. Elastic weight consolidation (EWC) [15] evaluates the importance of the parameters with fisher information. Other parameter importance evaluation methods have also been tried, like the entire learning trajectory in parameter space [35] or sensitivity of outputs and inputs [2]. Some methods replace parameter importance with group importance [1, 13]. One shortcoming of these methods is that model capacity is fixed, and the penalty loss will make the model's plasticity decrease with the increase of old tasks.

Memory-based Methods These methods keep a memory buffer for saving some information of old tasks. The usage of memory varies among different methods. Experience replay (ER) [7] uses memory to keep old samples and rehearses old samples to overcome CF when learning a new task. Some methods improve ER by replaying more disturbed old samples [3] or keeping diverse samples in memory [5]. Gradient episode memory (GEM) [20] and Average GEM (A-GEM) [6] also keep samples in memory and use old samples to estimate the gradients of old tasks. Saving real samples may raise privacy issues [17]. Gradient projection memory (GPM) [24] uses memory to maintain orthogonal bases and performs orthogonal projection to seek rectified updating direction. Some methods [19, 31] follow a similar idea to GPM and maintain a projection matrix for each layer. Some method [17] tries to get better plasticity-stability trade-off when rectifying gradient direction with projection operation. Trust region gradient projection (TRGP) [18] defines the trust region and leverages it to improve model's performance on new tasks. Flattening sharpness for dynamic gradient projection memory (FS-DGPM) [8] uses memory and new data to flatten the loss landscape and evaluate the importance of bases in GPM. Like regularization-based methods, all these methods also keep a fixed model capacity and the model's plasticity inevitably decreases with the increase of old tasks.

Expansion-based Methods These methods dynamically expand the model's architecture for each new coming task. Progressive neural network (PNN) [23] adds new sub-networks with connections for previous architecture and expands the parameters super-linearly. Some works, like calibrating CNNs for lifelong learning (CCLL) [27] and rectification-based knowledge retention (RKR) [26], expand an equal number of parameters for each new task. Some works, like additive parameter decomposition (APD) [33] and dynamic expand network (DEN) [34], use regularization terms to constrain the increase of expanded parameters. There are also some works [16, 28] defining a search space with different expansion strategies. When adding (expanding) additional parameters, all these methods do not consider how to evaluate the model's plasticity quantitatively and improve it adaptively.

3. Methodology

Figure 1 gives an illustration of our API method for a simple three-layer neural network. Except for the last layer, each layer can represent either a linear layer or a convolution layer, where each line represents a weight value in the linear layer or a kernel in the convolution layer. The blue part in Figure 1 is the original neural network, and we use $\boldsymbol{W}_l \in \mathbb{R}^{d_O^l \times d_I^l}$ to represent the weight of blue part in the l-th layer. Note that we omit the kernel dimensions in the convolution layer for simplicity. d_O^l and d_I^l represent the dimensions (channels) of the output and input, respectively. Besides the blue part, each layer may expand additional red part by increasing the input dimension d_I^l. Here, we use $d_I^{l,t}$ to denote the input dimension in the l-th layer when the model learns task t and use $\boldsymbol{W}_{l,t} \in \mathbb{R}^{d_O^l \times d_I^{l,t}}$ to denote the corresponding weight. Here, $d_I^{l,t} \geq d_I^{l,t-1}$ and $d_I^{l,1} = d_I^l$. $\boldsymbol{W}_{l,t}$ is expandable and includes both the blue part and expanded red part. In other words, $\boldsymbol{W}_{l,t-1} = \boldsymbol{W}_{l,t}[:, : d_I^{l,t-1}]$ and $\boldsymbol{W}_{l,1} = \boldsymbol{W}_l$. We will give the motivation of the API architecture in Section 3.3.

For each new task t, API first evaluates the model's plasticity with current parameters $\boldsymbol{W}_{l,t-1}$ when overcoming CF. Then, API adaptively expands $\boldsymbol{W}_{l,t-1}$ to $\boldsymbol{W}_{l,t}$ according to the evaluation results for improving the model's plasticity. Note that $\boldsymbol{W}_{l,t} = \boldsymbol{W}_{l,t-1}$ is possible, which means current

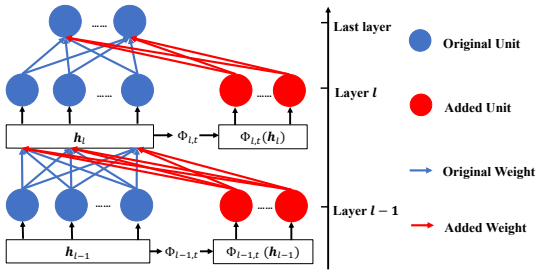


Figure 1. Illustration of API for a three-layer architecture. The blue part denotes the original architecture. The red part denotes the architecture for improving plasticity.

plasticity is enough for the model to learn the new task. Finally, API learns the new task with $\boldsymbol{W}_{l,t}$ when overcoming CF on $\boldsymbol{W}_{l,t-1}$.

API adopts the gradient rectification strategy to overcome CF. Methods based on this strategy rectify new task gradient so that it will not interfere with the model's performance on the old task. We will show that a representative gradient rectification method, GPM [24], suffers from constantly increasing memory usage (see Section 3.1). Thus, API proposes dual GPM (DualGPM) to overcome CF. DualGPM can achieve similar accuracy as GPM, but its memory usage does not increase all the time. Furthermore, based on DualGPM, API defines a new metric, called gradient retention ratio (GRR), to evaluate and improve the model's plasticity. The following subsections will describe the detail of the main components in API, including DualGPM, plasticity evaluation, and plasticity improvement.

3.1. Dual Gradient Projection Memory

We use $\mathcal{M}_{l,t}$ to denote the subspace containing the gradients of the previous $t-1$ old tasks for the l-th layer when the model learns task t ($1 \leq t \leq T$). We use $\mathcal{M}_{l,t}^{\perp}$ to denote the orthogonal complement of $\mathcal{M}_{l,t}$. This means:

$$\mathcal{M}_{l,t}^{\perp} = \{\boldsymbol{u}^{\perp} \in \mathbb{R}^{d_l} | \, \forall \, \boldsymbol{u} \in \mathcal{M}_{l,t}, \; (\boldsymbol{u}^{\perp})^T \boldsymbol{u} = 0\}, \quad (2)$$
$$\mathcal{M}_{l,t} \oplus \mathcal{M}_{l,t}^{\perp} = \mathbb{R}^{d_l}, \; \dim(\mathcal{M}_{l,t}) + \dim(\mathcal{M}_{l,t}^{\perp}) = d_l.$$

Here, d_l denotes the gradient dimension and $\oplus$ denotes direct sum in linear algebra [11]. Obviously, $\mathcal{M}_{l,1} = \{\boldsymbol{0}\}$ and $\mathcal{M}_{l,1}^{\perp} = \mathbb{R}^{d_l}$. According to the existing works [24, 31, 36], the following proposition holds:

Proposition 1. *The gradient update of linear or convolution layer lies in the span of inputs.*

Please refer to existing work [24] or supplementary material for the explanation of this proposition. With this proposition, $\mathcal{M}_{l,t}$ can be computed by finding the subspace containing the inputs of previous $t-1$ old tasks. The details of getting $\mathcal{M}_{l,t}$ are shown in the process of DualGPM (see Section 3.1.1).

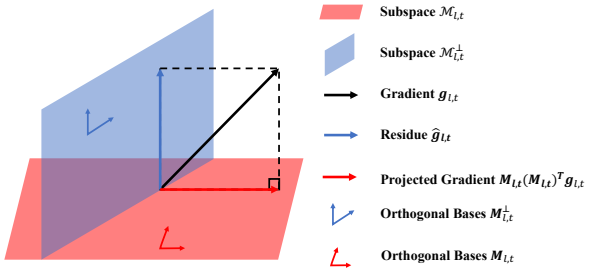


Figure 2. Illustration of orthogonal projection. Orthogonal projection projects the gradient into $\mathcal{M}_{l,t}$. GPM removes the projected component and makes the residue orthogonal to $\mathcal{M}_{l,t}$. Note that $\mathcal{M}_{l,t}$ contains the gradients of all previous tasks.

GPM [24] overcomes CF by orthogonal projection. Specifically, it maintains orthogonal bases of $\mathcal{M}_{l,t}$ and projects new task gradient $\boldsymbol{g}_{l,t}$ into $\mathcal{M}_{l,t}$ by $\boldsymbol{M}_{l,t}(\boldsymbol{M}_{l,t})^T\boldsymbol{g}_{l,t}$. Here, $\boldsymbol{M}_{l,t} = [\boldsymbol{u}_1, ..., \boldsymbol{u}_m]$ denotes orthogonal bases of $\mathcal{M}_{l,t}$ and $m = \dim(\mathcal{M}_{l,t})$. Then GPM removes the projected gradient from $\boldsymbol{g}_{l,t}$ by

$$\hat{\boldsymbol{g}}_{l,t} = \boldsymbol{g}_{l,t} - \boldsymbol{M}_{l,t}(\boldsymbol{M}_{l,t})^T\boldsymbol{g}_{l,t}. \quad (3)$$

Here, $\hat{\boldsymbol{g}}_{l,t}$ is the residue that lies in $\mathcal{M}_{l,t}^{\perp}$. Figure 2 gives an illustration of orthogonal projection. Since $\dim(\mathcal{M}_{l,t})$ increases with the number of tasks, the memory usage of GPM for storing $\boldsymbol{M}_{l,t}$ also increases with the number of tasks. We propose DualGPM, which achieves orthogonal projection with memory not increasing all the time. In the following discussion, we first show how DualGPM works in the layers with non-expandable $\boldsymbol{W}_{l,t}$ ($d_I^{l,t} \equiv d_I^l$). Then, we extend DualGPM to the layer with expandable $\boldsymbol{W}_{l,t}$.

3.1.1 Layers with Non-Expandable Parameters

Different from GPM which maintains orthogonal bases of $\mathcal{M}_{l,t}$, DualGPM maintains either orthogonal bases of $\mathcal{M}_{l,t}$ or orthogonal bases of $\mathcal{M}_{l,t}^{\perp}$ to perform orthogonal projection. When keeping orthogonal bases of $\mathcal{M}_{l,t}$ in memory, DualGPM uses operation (3) like GPM. When keeping orthogonal bases of $\mathcal{M}_{l,t}^{\perp}$ in memory, DualGPM performs orthogonal projection through

$$\hat{\boldsymbol{g}}_{l,t} = \boldsymbol{M}_{l,t}^{\perp}(\boldsymbol{M}_{l,t}^{\perp})^T\boldsymbol{g}_{l,t}. \quad (4)$$

Here, $\boldsymbol{M}_{l,t}^{\perp} = [\boldsymbol{u}_1^{\perp}, ..., \boldsymbol{u}_z^{\perp}]$ denotes orthogonal bases of $\mathcal{M}_{l,t}^{\perp}$ and $z = \dim(\mathcal{M}_{l,t}^{\perp})$. Note that operation (3) and operation (4) are equivalent and we call them dual operations.

DualGPM decides whether to keep $\boldsymbol{M}_{l,t}$ or $\boldsymbol{M}_{l,t}^{\perp}$ in memory according to $\dim(\mathcal{M}_{l,t})$ and $\dim(\mathcal{M}_{l,t}^{\perp})$. Specifically, during the learning of the first several tasks, $\dim(\mathcal{M}_{l,t}) \leq \dim(\mathcal{M}_{l,t}^{\perp})$. At this time, DualGPM maintains $\boldsymbol{M}_{l,t}$, and expands $\boldsymbol{M}_{l,t}$ to $\boldsymbol{M}_{l,t+1}$ after each task. When $\dim(\mathcal{M}_{l,t})$ increases and exceeds $\dim(\mathcal{M}_{l,t}^{\perp})$, DualGPM obtains $\boldsymbol{M}_{l,t}^{\perp}$ through some transformations on $\boldsymbol{M}_{l,t}$.

After that, DualGPM only maintains $\boldsymbol{M}_{l,t}^{\perp}$ in memory, and reduces $\boldsymbol{M}_{l,t}^{\perp}$ to $\boldsymbol{M}_{l,t+1}^{\perp}$ after each task. Through this way, the number of bases kept for each layer is $\min\{\dim(\mathcal{M}_{l,t}), \dim(\mathcal{M}_{l,t}^{\perp})\}$.

To make DualGPM work, we have to solve the following three key problems: expanding the bases of $\mathcal{M}_{l,t}$, obtaining the bases of $\mathcal{M}_{l,t}^{\perp}$ through the bases of $\mathcal{M}_{l,t}$, and reducing the bases of $\mathcal{M}_{l,t}^{\perp}$.

Expanding the Bases of $\mathcal{M}_{l,t}$ The expansion of $\mathcal{M}_{l,t}$ is the same as that in GPM. Specifically, according to Proposition 1, expanding the bases of $\mathcal{M}_{l,t}$ is equivalent to expanding the bases of input space. DualGPM computes the inputs matrix $\boldsymbol{R}_{l,t}$ such that each column of $\boldsymbol{R}_{l,t}$ represents an input of this layer. Getting the input matrix for convolution layer requires reshaping operation. Please refer to GPM [24] or supplementary material for details. Then, the part of $\boldsymbol{R}_{l,t}$ that has already in $\mathcal{M}_{l,t}$ is removed by

$$\hat{\boldsymbol{R}}_{l,t} = \boldsymbol{R}_{l,t} - \boldsymbol{M}_{l,t}(\boldsymbol{M}_{l,t})^T \boldsymbol{R}_{l,t} = \boldsymbol{R}_{l,t} - \boldsymbol{R}_{l,t,proj}. \quad (5)$$

Please note that when $t = 1$, $\dim(\mathcal{M}_{l,t}) = 0$ and hence $\boldsymbol{R}_{l,t,proj}$ is a zero matrix. After that, singular value decomposition (SVD) is performed on $\hat{\boldsymbol{R}}_{l,t} = \hat{\boldsymbol{U}}\hat{\boldsymbol{\Sigma}}\hat{\boldsymbol{V}}^T$. Then, u new orthogonal bases are chosen from the columns of $\hat{\boldsymbol{U}}$ for a minimum of u satisfying the following criteria for given threshold ϵ_{th}^{l}:

$$||(\hat{\boldsymbol{R}}_{l,t})_u||_F^2 + ||\boldsymbol{R}_{l,t,proj}||_F^2 \geq \epsilon_{th}^{l}||\boldsymbol{R}_{l,t}||_F^2. \quad (6)$$

Here, $(\hat{\boldsymbol{R}}_{l,t})_u = [\boldsymbol{u}_1, ..., \boldsymbol{u}_u]$ denotes the components of $\hat{\boldsymbol{R}}_{l,t}$ that correspond to top-u singular values. Then, subspace $\mathcal{M}_{l,t+1}$ is obtained with the bases $\boldsymbol{M}_{l,t+1} = [\boldsymbol{M}_{l,t}, \boldsymbol{u}_1, ..., \boldsymbol{u}_u]$.

Transforming $\mathcal{M}_{l,t}$ to $\mathcal{M}_{l,t}^{\perp}$ DualGPM transforms $\mathcal{M}_{l,t}$ to $\mathcal{M}_{l,t}^{\perp}$ by performing SVD to the matrix $\boldsymbol{M}_{l,t}$. Specifically, let $\boldsymbol{M}_{l,t} = \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^T$, we can prove that the column vectors of $\boldsymbol{U}$ which correspond to the zero singular values form a set of orthogonal bases of $\mathcal{M}_{l,t}^{\perp}$. We give the proof in supplementary material.

Reducing the Bases of $\mathcal{M}_{l,t}^{\perp}$ DualGPM reduces space $\mathcal{M}_{l,t}^{\perp}$ by removing the part of $\mathcal{M}_{l,t}^{\perp}$ which contains the gradient of the t-th task. Specifically, DualGPM first computes the input matrix $\boldsymbol{R}_{l,t}$. Then, the part of $\boldsymbol{R}_{l,t}$ which lies in $\mathcal{M}_{l,t}^{\perp}$ can be computed through

$$\hat{\boldsymbol{R}}_{l,t}^{\perp} = \boldsymbol{M}_{l,t}^{\perp}(\boldsymbol{M}_{l,t}^{\perp})^T \boldsymbol{R}_{l,t} = \boldsymbol{R}_{l,t,proj}^{\perp}. \quad (7)$$

After that, SVD is performed on $\hat{\boldsymbol{R}}_{l,t}^{\perp} = \hat{\boldsymbol{U}}^{\perp}\hat{\boldsymbol{\Sigma}}^{\perp}(\hat{\boldsymbol{V}}^{\perp})^T$. Then, k new orthogonal bases are chosen from the columns of $\hat{\boldsymbol{U}}^{\perp}$ for a maximum of k satisfying the following criteria for the given threshold ϵ_{th}^{l} (the same as ϵ_{th}^{l} in (6)):

$$||(\hat{\boldsymbol{R}}_{l,t}^{\perp})_k||_F^2 \leq (1 - \epsilon_{th}^{l})||\boldsymbol{R}_{l,t}||_F^2. \quad (8)$$

Let $\boldsymbol{Z} = (\hat{\boldsymbol{R}}_{l,t}^{\perp})_k = [\boldsymbol{u}_1^{\perp}, ..., \boldsymbol{u}_k^{\perp}]$, $\mathcal{Z} = \text{span}\{\boldsymbol{u}_1^{\perp}, ..., \boldsymbol{u}_k^{\perp}\}$. Here, $\mathcal{Z}$ is the subspace of $\mathcal{M}_{l,t}^{\perp}$ that contains the gradient of the t-th task. DualGPM removes $\mathcal{Z}$ from $\mathcal{M}_{l,t}^{\perp}$ to get $\mathcal{M}_{l,t+1}^{\perp}$. Specifically, let $\hat{\boldsymbol{M}}_{l,t}^{\perp} = \boldsymbol{M}_{l,t}^{\perp} - \boldsymbol{Z}(\boldsymbol{Z}^T)\boldsymbol{M}_{l,t}^{\perp}$. DualGPM performs the second SVD on $\hat{\boldsymbol{M}}_{l,t}^{\perp} = \widetilde{\boldsymbol{U}}^{\perp}\widetilde{\boldsymbol{\Sigma}}^{\perp}(\widetilde{\boldsymbol{V}}^{\perp})^T$. We can prove that the columns of $\widetilde{\boldsymbol{U}}^{\perp}$ which correspond to the non-zero singular values form the bases $\boldsymbol{M}_{l,t+1}^{\perp}$. We give the proof in supplementary material.

Comparing DualGPM with GPM DualGPM considers both $\mathcal{M}_{l,t}$ and $\mathcal{M}_{l,t}^{\perp}$. Therefore, DualGPM keeps $\min\{\dim(\mathcal{M}_{l,t}), \dim(\mathcal{M}_{l,t}^{\perp})\}$ bases in memory for each layer. Different from DualGPM, GPM only considers the space $\mathcal{M}_{l,t}$ and keeps $\dim(\mathcal{M}_{l,t})$ bases in memory for each layer. Since $\dim(\mathcal{M}_{l,t})$ increases and $\dim(\mathcal{M}_{l,t}^{\perp})$ decreases with the increase of t, DualGPM keeps much fewer bases than GPM when t is large. Note that updating bases in memory only happens after each task, and hence DualGPM does not cause too much computation for SVD operations. Section 4 will show that DualGPM gets similar performance to GPM and uses much less memory than GPM.

3.1.2 Layers with Expandable Parameters

In the layers with expandable $\boldsymbol{W}_{l,t}$, updating memory bases (see (5) and (7)) cannot be performed directly since the dimension of the inputs in $\boldsymbol{R}_{l,t}$ may be higher than that of the bases in $\boldsymbol{M}_{l,t}$. Based on the fact that any d-dimensional vector $\boldsymbol{g} = [g_1, ..., g_d]^T$ can be embedded into a higher dimensional space by $\boldsymbol{g} \leftarrow [g_1, g_2, ..., g_d, 0, ..., 0]^T$, we can embed $\boldsymbol{M}_{l,t}$ into the space which the gradient of the new task lies in. $\boldsymbol{M}_{l,t}^{\perp}$ can also be obtained through (2). Mathematically, new $\boldsymbol{M}_{l,t}$ and $\boldsymbol{M}_{l,t}^{\perp}$ are got by

$$\boldsymbol{M}_{l,t} \leftarrow \begin{bmatrix} \boldsymbol{M}_{l,t} \\ \boldsymbol{O} \end{bmatrix}, \quad \boldsymbol{M}_{l,t}^{\perp} \leftarrow \begin{bmatrix} \boldsymbol{M}_{l,t}^{\perp} & \boldsymbol{O} \\ \boldsymbol{O} & \boldsymbol{I} \end{bmatrix}, \quad (9)$$

where $\boldsymbol{O}$ denotes zero matrix and $\boldsymbol{I}$ denotes identity matrix. After the operation in (9), we can update memory according to the description in Section 3.1.1.

Algorithm 1 shows the process of DualGPM, including the case of non-expandable parameters and the case of expandable parameters.

3.2. Plasticity Evaluation

DualGPM constrains the new task gradient $\boldsymbol{g}_{l,t}$ in the subspace $\mathcal{M}_{l,t}^{\perp}$ (see (3) and (4)). We define a metric called gradient retention ratio (GRR) for evaluating the constraint. The GRR of the l-th neural network layer for task t can be computed as

$$\text{GRR}(l,t) = E_{\boldsymbol{x}\sim\mathcal{D}_t}\left[\frac{||(\hat{\boldsymbol{g}}_{l,t})_{\boldsymbol{x}}||_2}{||(\boldsymbol{g}_{l,t})_{\boldsymbol{x}}||_2}\right], \quad (10)$$

Algorithm 1 DualGPM

Input: Current task data $\mathcal{D}_t$, a neural network model $f(\cdot, \boldsymbol{\Theta})$ with L layers, $\boldsymbol{\Theta} = \{\boldsymbol{W}_{l,t}\}_{l=1}^{L}$, orthogonal bases memory $\{\boldsymbol{M}_{l,t-1}^{*}\}_{l=1}^{L}$.
Output: Updated orthogonal bases memory $\{\boldsymbol{M}_{l,t}^{*}\}_{l=1}^{L}$.
Get input matrix $\{\boldsymbol{R}_{l,t}\}_{l=1}^{L}$ through $\mathcal{D}_t$ and $f(\cdot, \boldsymbol{\Theta})$;
for l in $1 : L$ **do**
  if $\boldsymbol{M}_{l,t-1}^{*}$ is $\boldsymbol{M}_{l,t-1}$ **then**
    Embed $\boldsymbol{M}_{l,t-1}$ into higher dimensional space by (9);
    // Only for the layers with expandable parameters
    Expand $\boldsymbol{M}_{l,t-1}$ to $\boldsymbol{M}_{l,t}$ by (5) and (6);
    if $\dim(\mathcal{M}_{l,t}) > \dim(\mathcal{M}_{l,t}^{\perp})$ **then**
      Transform matrix $\boldsymbol{M}_{l,t}$ to matrix $\boldsymbol{M}_{l,t}^{\perp}$ through SVD;
    end if
  else if $\boldsymbol{M}_{l,t-1}^{*}$ is $\boldsymbol{M}_{l,t-1}^{\perp}$ **then**
    Embed $\boldsymbol{M}_{l,t-1}^{\perp}$ into higher dimensional space by (9);
    // Only for the layers with expandable parameters
    Reduce $\boldsymbol{M}_{l,t-1}^{\perp}$ to $\boldsymbol{M}_{l,t}$ by (7) and (8);
    if $\dim(\mathcal{M}_{l,t}) < \dim(\mathcal{M}_{l,t}^{\perp})$ **then**
      Transform matrix $\boldsymbol{M}_{l,t}^{\perp}$ to matrix $\boldsymbol{M}_{l,t}$ through SVD;
    end if
  end if
end for

where $(\boldsymbol{g}_{l,t})_{\boldsymbol{x}}$ represents the gradient in this layer with input sample $\boldsymbol{x}$. $\hat{\boldsymbol{g}}_{l,t}$ is obtained by (3) or (4). In Equation (10), ratio $\frac{||\hat{\boldsymbol{g}}_{l,t}||_2}{||\boldsymbol{g}_{l,t}||_2}$ is smaller than 1 due to the orthogonal projection. The smaller the value of $\frac{||\hat{\boldsymbol{g}}_{l,t}||_2}{||\boldsymbol{g}_{l,t}||_2}$ is, the larger the part of gradient is removed by (3) or (4). In an extreme case where $\dim(\mathcal{M}_{l,t}) = d_l$ and $\dim(\mathcal{M}_{l,t}^{\perp}) = 0$, $\hat{\boldsymbol{g}}_{l,t}$ is always $\mathbf{0}$. This means the parameters of this layer can not be updated for learning new task t. In other words, this layer has no plasticity. We further use $\text{AGRR}(t) = \frac{1}{L}\sum_{l=1}^{L}\text{GRR}(l,t)$ to denote the average GRR of all layers, where L denotes the number of layers. AGRR evaluates the average constraint caused by DualGPM.

Then, we show the relation between AGRR and the model's performance. We perform DualGPM on Split CIFAR100, which is a popular continual learning dataset we use for experiments in Section 4. We vary the threshold ϵ_{th}^{l} in (6) and (8). Obviously, larger ϵ_{th}^{l} makes $\dim(\mathcal{M}_{l,t})$ larger, and thus makes AGRR smaller. Figure 3 (a) shows the relation between AGRR and the average gradient norm $\frac{1}{S}\sum_{i=1}^{S}||\hat{\boldsymbol{g}}_{i,2}||_2$ for learning task 2. Here, S denotes the number of times the model updates the parameters when learning task 2. From Figure 3 (a), we can find that average gradient norm decreases with the decrease of AGRR. This means the model changes less and less for learning the new task. Since plasticity describes the model's ability to change itself [21], the model's plasticity decreases with the decrease of AGRR. Figure 3 (b) shows the relation between AGRR and accuracy on task 2 when the learning of task 2 is over. From this figure, we can find that model's accuracy

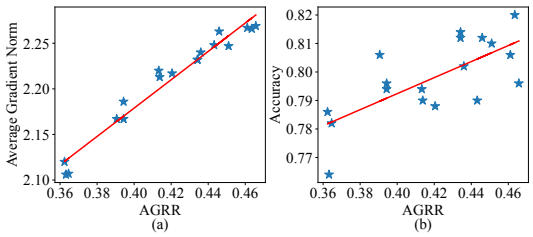


Figure 3. DualGPM with non-expandable parameters learns on Split CIFAR100. (a) shows the correlation between AGRR and average gradient norm for learning task 2. (b) shows the correlation between AGRR and accuracy on task 2.

also decreases with the decrease of AGRR. From these results, we can find that AGRR has a high correlation with the model's performance and the model's ability to change. Therefore, API uses AGRR to evaluate the plasticity of the model.

3.3. Plasticity Improvement

In Section 3.2, we have shown that metric AGRR can evaluate the constraint caused by DualGPM. We also show that AGRR has a high correlation with the model's performance and the model's ability to change. Therefore, API tries to increase AGRR to improve the model's plasticity. According to Proposition 1, increasing GRR of the l-th layer can be achieved by increasing the input dimension d_I^l. Hence, API improves plasticity by increasing the input dimension

With GRR, the input dimension $d_I^{l,t}$ is decided as

$$d_I^{l,t} = d_I^{l,t-1} + \max\left(\lfloor K(\rho - \text{GRR}(l,t)) + 0.5 \rfloor, 0\right), \tag{11}$$

where $\lfloor \cdot \rfloor$ denotes round down. K and ρ are hyperparameters. For all the experiments, we set K and ρ as 10 and 0.5, unless otherwise stated. Note that when $\text{GRR}(l,t) \geq \rho$, $d_I^{l,t} = d_I^{l,t-1}$ and no new parameters are added. With Equation (11), we try to give larger expansion to the layer with smaller GRR so that AGRR does not decrease too much with the increase of tasks.

After expanding $\boldsymbol{W}_{l,t-1}$ to $\boldsymbol{W}_{l,t}$ through (11), API increases the dimension of the input $\boldsymbol{h}_l$ through a transformation $\Phi_{l,t}$, where $\Phi_{l,t}(\boldsymbol{h}_l) = \boldsymbol{B}_{l,t} \bullet \boldsymbol{h}_l$, and $\widetilde{\boldsymbol{h}}_{l,t} = \text{Concat}(\boldsymbol{h}_l, \Phi_{l,t}(\boldsymbol{h}_l))$. Here $\boldsymbol{B}_{l,t} \in \mathbb{R}^{d_I^l \times n}$ is trainable parameters and $n = d_I^{l,t} - d_I^l$. Operation $\bullet$ denotes channel-wise combination in the convolution layer and dimension-wise combination in the linear layer. 'Concat' denotes the concatenation of the input dimension. Then, the forward propagation for the new task t in this layer can be computed as $\boldsymbol{h}_{l+1} = \sigma(\boldsymbol{W}_{l,t} * \widetilde{\boldsymbol{h}}_{l,t} + \boldsymbol{b})$, where σ is the activation function.

During the learning of task t, the part of $\boldsymbol{B}_{l,t}$ corresponding to the previous $t-1$ task is frozen to overcome CF.

Algorithm 2 The Whole Process of API

Input: The data of different tasks $\{\mathcal{D}_t\}_{t=1}^{T}$, a neural network model $f(\cdot, \mathbf{\Theta})$ with L layers, $\mathbf{\Theta} = \{\boldsymbol{W}_{l,1}\}_{l=1}^{L}$.
Output: Learned network $f(\cdot, \mathbf{\Theta})$ with $\mathbf{\Theta} = \{\boldsymbol{W}_{l,T}\}_{l=1}^{L}$.
Initialize orthogonal bases memory $\{\boldsymbol{M}_{l,1}^{*}\}_{l=1}^{L}$: $\boldsymbol{M}_{l,1}^{*} = \boldsymbol{M}_{l,1} = [\,]$;
Learn the neural network with the first dataset $\mathcal{D}_1$;
Update the memory $\{\boldsymbol{M}_{l,1}^{*}\}_{l=1}^{L}$ and get $\{\boldsymbol{M}_{l,2}^{*}\}_{l=1}^{L}$; // Refer to Algorithm 1
for t in $2:T$ **do**
  Compute $\{\mathrm{GRR}(l,t)\}_{l=1}^{L}$ by (10) for plasticity evaluation;
  Compute $\{d_I^{l,t}\}_{l=1}^{L}$ by (11) and expand $\boldsymbol{W}_{l,t-1}$ to $\boldsymbol{W}_{l,t}$ for plasticity improvement;
  for $ep = 1, 2, ..., num_{epochs}$ **do**
    for $\mathcal{B}_t$ sampled from $\mathcal{D}_t$ **do**
      Compute the loss $L(\mathcal{B}_t; \mathbf{\Theta})$ over $\mathcal{B}_t$ and get gradient $\boldsymbol{g}_t = [\boldsymbol{g}_{1,t}, \boldsymbol{g}_{2,t}, ..., \boldsymbol{g}_{L,t}]$;
      Using $\boldsymbol{M}_{l,t}^{*}$ to project gradient $\boldsymbol{g}_{l,t}$ by (3) or (4) and get $\hat{\boldsymbol{g}}_{l,t}$; // Orthogonal projection
      Update the parameters with projected gradient $\hat{\boldsymbol{g}}_t = [\hat{\boldsymbol{g}}_{1,t}, \hat{\boldsymbol{g}}_{2,t}, ..., \hat{\boldsymbol{g}}_{L,t}]$;
    end for
  end for
  Update the memory $\{\boldsymbol{M}_{l,t}^{*}\}_{l=1}^{L}$ and get $\{\boldsymbol{M}_{l,t+1}^{*}\}_{l=1}^{L}$; // Refer to Algorithm 1
end for

The part of $\boldsymbol{B}_{l,t}$ corresponding to only new task t is trained with $\boldsymbol{W}_{l,t}$ together. In the inference phase, for any task t ($t < T$), only $\boldsymbol{W}_{l,t}$ is used to perform prediction. The experiments in Section 4 will show that the expansion of $\boldsymbol{W}_{l,t}$ is limited.

In Algorithm 2, we give the whole process of API to show how the different components of API work together.

4. Experiment

4.1. Experimental Setup

Datasets We evaluate continual learning methods on four widely used datasets, including Split CIFAR100 [20], CIFAR100-sup [24], Split Mini-Imagenet, and 5-Datasets [9]. Split CIFAR100 is constructed by splitting 100 classes of CIFAR100 into 20 tasks, and each task consists of 5 exclusive classes. CIFAR100-sup has 20 tasks, each with 5 classes. The classes in each task of CIFAR100-sup come from the same superclass of CIFAR100. Split Mini-Imagenet is constructed by splitting 100 classes of Mini-Imagenet into 20 tasks, and each task consists of 5 classes. 5-Datasets is a continual learning benchmark with 5 different datasets, including CIFAR10, MNIST, SVHN, notMNIST, and Fashion-MNIST.

Baselines and Metrics For regularization-based methods, we compare with elastic weight consolidation (EWC) [15], adaptive group sparsity based continual learning (AGS-CL) [13] and active forgetting with synaptic expansion-convergence (AFEC) [30]. For memory-based methods, we compare with experience replay with reservoir sampling (ER-Res) [7], gradient episode memory (GEM) [20], gradient projection memory (GPM) [24], flattening sharpness dynamic gradient projection memory (FS-DGPM) [8], trust region gradient projection (TRGP) [18] and Connector [17]. For expansion-based methods, we compare with dynamic expansion network (DEN) [34], reinforcement continual learning (RCL) [32], additive parameter decomposition (APD) [33], calibrating CNNs for lifelong learning (CCLL) [27], and rectification-based retention (RKR) [26].

Following existing works [8,24], we use average final accuracy (ACC) and backward transfer (BWT) as evaluation metrics. ACC is the average accuracy of all tasks. BWT measures forgetting. The formulas of these two metrics are as follows

$$\mathrm{ACC} = \frac{1}{T}\sum_{i=1}^{T}\mathrm{ACC}_{T,i},$$
$$\mathrm{BWT} = \frac{1}{T-1}\sum_{i=1}^{T-1}(\mathrm{ACC}_{T,i} - \mathrm{ACC}_{i,i}). \quad (12)$$

Here, T is the total number of tasks and $\mathrm{ACC}_{j,i}$ is the model's accuracy on the i-th task after learning the j-th task. We also evaluate the memory usage for different methods.

Architectures and Training Details Following the existing works [24, 25], we use a 5-layer AlexNet for Split CIFAR100 and use a modified LeNet for CIFAR100-sup. For Split Mini-Imagenet and 5-Datasets, we use a reduced ResNet18 architecture like that in [4, 24].

Following GPM [24], we use stochastic gradient descent (SGD) to train all the architectures in all the experiments. Each task is trained for 200 epochs on Split CIFAR100, 50 epochs on CIFAR100-sup, 10 epochs on Split Mini-Imagenet, and 100 epochs on 5-Datasets to keep consistent with experimental settings in existing works [24]. For Split CIFAR100, CIFAR100-sup, and 5-Datasets, an early stopping strategy is applied. The batch size is set to be 64 for all the datasets to follow the existing work [24]. Since our DualGPM is an improvement of GPM, we set the value of threshold ϵ_{th}^{l} (see Equations (6) and (8)) for each layer to be consistent with GPM. We perform all experiments on four NVIDIA TITAN Xp GPUs.

4.2. Results

4.2.1 Accuracy

We repeat all the experiments five times with different random seeds. Table 1 shows the comparison of our API with

Table 1. Results of different continual learning methods on four datasets.

Methods	CIFAR100-sup		Split CIFAR100		Split Mini-Imagenet		5-Datasets	
	ACC (%)	BWT (%)	ACC (%)	BWT (%)	ACC (%)	BWT (%)	ACC (%)	BWT (%)
EWC [15]	46.7 ± 0.6	-13.5 ± 1.1	75.3 ± 0.7	-6.3 ± 0.6	52.1 ± 1.1	-9.3 ± 1.4	84.3 ± 0.2	-2.1 ± 0.2
AGS-CL [13]	56.3 ± 2.9	-2.3 ± 2.0	76.2 ± 0.4	-3.0 ± 0.3	55.1 ± 0.9	-1.5 ± 0.4	86.2 ± 0.4	-3.5 ± 0.3
AFEC [30]	56.2 ± 1.4	-6.2 ± 1.4	78.7 ± 0.5	-2.5 ± 0.4	57.6 ± 0.6	-2.0 ± 1.2	88.6 ± 0.3	-1.8 ± 0.3
ER-Res [7]	53.3 ± 0.7	-3.4 ± 0.8	79.2 ± 0.4	-4.9 ± 0.5	55.2 ± 2.9	-5.7 ± 0.8	83.4 ± 0.7	-8.6 ± 0.9
GEM [20]	50.4 ± 0.9	-7.4 ± 0.7	77.9 ± 0.2	-6.4 ± 0.5	-	-	-	-
FS-DGPM [8]	58.5 ± 0.6	-4.0 ± 0.6	80.5 ± 0.4	-3.3 ± 0.4	-	-	-	-
Connector [17]	56.2 ± 0.3	-0.4 ± 0.3	78.1 ± 0.2	-0.3 ± 0.2	57.8 ± 0.8	2.1 ± 0.1	85.5 ± 0.3	-2.9 ± 0.5
GPM [24]	57.7 ± 0.7	-1.2 ± 0.4	78.9 ± 0.2	-0.1 ± 0.2	61.2 ± 0.6	**0.3 ± 0.3**	88.8 ± 0.6	-2.0 ± 0.3
TRGP [18]	58.2 ± 0.2	-1.7 ± 0.5	80.5 ± 0.3	-0.3 ± 0.2	62.5 ± 0.7	-0.2 ± 0.4	90.9 ± 0.1	**-0.1 ± 0.0**
DualGPM	57.6 ± 0.7	-1.0 ± 0.2	78.5 ± 0.4	**-0.0 ± 0.3**	61.2 ± 0.6	**0.3 ± 0.4**	88.7 ± 0.5	-1.9 ± 0.2
API	**60.2 ± 0.2**	**-0.2 ± 0.1**	**81.4 ± 0.4**	-0.8 ± 0.2	**65.9 ± 0.6**	-0.3 ± 0.2	**91.1 ± 0.3**	-0.5 ± 0.1

Table 2. The performance of different expansion-based methods on CIFAR100-sup dataset.

Methods	DEN [34]	RCL [32]	APD [33]	CCLL [27]	RKR [26]	GPM [24]	API
Accuracy (%)	51.10	51.99	56.81	55.2	58.3	57.7	**60.2**
Capacity (%)	191	184	130	106	116	**100**	105

memory-based and regularization-based methods. DualGPM denotes a variant of our method with fixed model capacity and without adaptive improvement component. We can find that DualGPM achieves similar accuracy as GPM. Please note that DualGPM uses much less memory than GPM, which will be verified in the following subsection. API achieves the best results on all datasets. EWC, AGS-CL, AFEC, ER-Res, GEM, and FS-DGPM suffer from CF. For example, GEM achieves 77.9% in accuracy and 6.4% in forgetting on Split CIFAR100. This means if GEM has no forgetting, its accuracy is 84.3%.

TRGP, GPM, and our API show better performance in overcoming CF than other methods. Among these, GPM achieves 78.9% in accuracy and 0.1% in forgetting on Split CIFAR100. This means that even if there is no forgetting in GPM, the accuracy of GPM can only reach 79.0%, which is still lower than our API method. Similar phenomena also happen on other datasets. Figure 4 shows relative accuracy improvement on Split CIFAR100 and Split Mini-Imagenet, where relative accuracy improvement is the accuracy of API or TRGP minus the accuracy of GPM. We can find that both API and TRGP improve over GPM on most tasks, and our API shows a larger improvement than TRGP. Furthermore, the improvement of our API has an increasing trend with the increase of tasks. This is because as the number of tasks increases, the plasticity of the GPM gradually decreases. Our method API keeps improving the plasticity of the model. Therefore, as the task increases, our method API shows larger and larger improvement over GPM.

We also follow existing works [18, 24] and compare our API with many expansion-based methods on CIFAR100-sup. The results are shown in Table 2. Here, capacity [33] denotes $\frac{|\mathbf{\Theta}_0|}{|\mathbf{\Theta}_T|}$, where $|\mathbf{\Theta}_0|$ is the number of parameters before the first task and $|\mathbf{\Theta}_T|$ is the number of parameters after the last task. GPM uses a fixed-size network and its capacity is always 100%. API and expansion-based methods require additional parameters during the training, and their capacities are larger than GPM. However, API gets a smaller capacity and better accuracy than expansion-based methods.

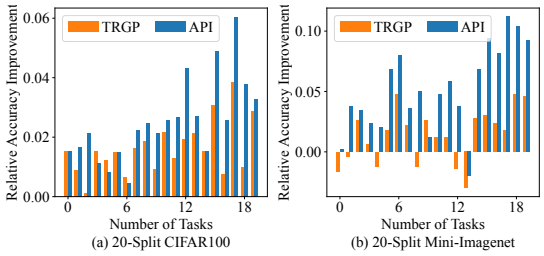


Figure 4. Relative accuracy improvement for different methods. Relative accuracy improvement is the accuracy of API or TRGP minus the accuracy of GPM.

4.2.2 Memory Usage

We compare memory usage for different methods. We focus on the methods that do not save real samples in memory since these methods do not raise privacy concerns.

Figure 5 (a) shows the variation of the saved bases in the third layer of AlexNet on the experiment of Split CIFAR100. We can find that the number of bases stored by GPM increases all the time since GPM only considers $\mathcal{M}_{l,t}$. Our methods API and DualGPM consider both $\mathcal{M}_{l,t}$ and $\mathcal{M}_{l,t}^{\perp}$. Therefore, the bases stored by our methods increase first and then decrease. Furthermore, the bases stored by API are more than the bases stored by DualGPM. This is because API expands parameters, which may increase the

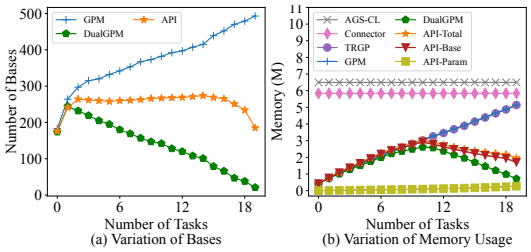


Figure 5. (a) Variation of saved bases in the third layer of AlexNet when the model learns on Split CIFAR100. (b) Variation of whole memory usage for different methods on Split CIFAR100.

Table 3. The performance for different methods on Split CIFAR100 dataset and 5-Datasets. MEM denotes the memory usage for saving bases and expanded parameters.

	Split CIFAR100		5-Datasets	
Methods	ACC (%)	MEM (M)	ACC (%)	MEM (M)
API (GPM)	81.2 ± 0.2	7.3	$\mathbf{91.1 \pm 0.2}$	7.7
API	$\mathbf{81.4 \pm 0.4}$	**2.0**	$\mathbf{91.1 \pm 0.3}$	**3.1**

number of bases (see (9)).

Figure 5 (b) gives the variation of memory usage on Split CIFAR100. API-Base denotes the memory for storing bases. API-Param denotes memory for expanding parameters. API-Total denotes the sum of API-Base and API-Param. We can find that our methods use the least memory among all the methods. Furthermore, GPM's memory usage increases all the time. However, API-Total and API-Base increase first and then decrease. API-Param increases all the time, but it is much less than API-Base.

4.2.3 Ablation Study

We replace DualGPM with GPM in API and give the results in Table 3. Here API (GPM) denotes the variant of API that uses GPM in API for overcoming CF. API is our original method that uses DualGPM for overcoming CF. We can find that API (GPM) performs similarly to API but uses much more memory.

To verify the effectiveness of using (11) for plasticity improvement, we replace GRR with a constant value. This means the model adds an equal number of channels for each layer before learning each new task and we call this strategy 'Equal'. We use C to denote the number of added channels for each task in 'Equal' and vary C in $[1, 2, 3]$. Obviously, increasing C will increase the expanded parameters and thus increase memory usage.

Table 4 shows the expanded parameters and accuracy for each experiment. We can find that 'Equal' gets better results when expanding more parameters. However, when getting similar accuracy to API, 'Equal' requires more parameters to improve the model's plasticity. This shows the superiority of using (11) for improvement.

Table 4. Performance of different expansion strategies on Split CIFAR100 and 5-Datasets. Param denotes the number of expanded parameters.

	Split CIFAR100		5-Datasets	
Methods	ACC (%)	Param (M)	ACC (%)	Param (M)
Equal (C=1)	79.5 ± 0.3	**0.20**	90.3 ± 0.2	**0.06**
Equal (C=2)	80.5 ± 0.4	0.40	90.7 ± 0.4	0.12
Equal (C=3)	$\mathbf{81.4 \pm 0.3}$	0.60	90.9 ± 0.2	0.19
API	$\mathbf{81.4 \pm 0.4}$	0.26	$\mathbf{91.1 \pm 0.3}$	0.11

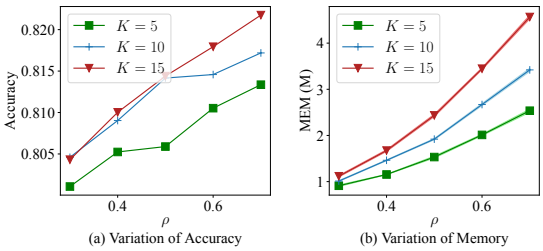


Figure 6. Accuracy and memory usage with different hyperparameters. Here, memory usage (MEM) is the memory for saving bases and expanded parameters.

4.2.4 Hyperparameter Analysis

We vary the value of ρ and K in (11). Specifically, ρ is varied in $[0.3, 0.4, 0.5, 0.6, 0.7]$ and K is varied in $[5, 10, 15]$. Figure 6 (a) shows API's accuracy on Split CIFAR100. Figure 6 (b) shows API's memory usage on Split CIFAR100. We can find that both API's accuracy and memory usage increase with the increase of ρ and K. This is intuitively reasonable since increasing ρ and K makes the model expand more parameters and thus give larger improvement in plasticity. We choose $\rho = 0.5$ and $K = 10$ to make a better trade-off between memory and accuracy.

5. Conclusion

In this work, we propose a new method, called API, for continual learning. Besides the ability to overcome catastrophic forgetting (CF), API evaluates a model's plasticity and improves plasticity adaptively for a new task if necessary. Experiments in the task incremental setting, where task identities are available for testing, show that API can achieve better performance than other state-of-the-art baselines. Future work will extend API to other continual learning settings, like those where task identities are unavailable for testing.

Acknowledgment

This work is supported by NSFC (No.62192783), National Key R&D Program of China (No.2020YFA0713901) and NSFC (No.61921006).

References

[1] Hongjoon Ahn, Sungmin Cha, Donggyu Lee, and Taesup Moon. Uncertainty-based continual learning with adaptive regularization. In *Advances in Neural Information Processing Systems*, pages 4392–4402, 2019. 2

[2] Rahaf Aljundi, Francesca Babiloni, Mohamed Elhoseiny, Marcus Rohrbach, and Tinne Tuytelaars. Memory aware synapses: learning what (not) to forget. In *European Conference on Computer Vision*, pages 139–154, 2018. 2

[3] Rahaf Aljundi, Eugene Belilovsky, Tinne Tuytelaars, Laurent Charlin, Massimo Caccia, Min Lin, and Lucas Page-Caccia. Online continual learning with maximal interfered retrieval. In *Advances in Neural Information Processing Systems*, pages 11849–11860, 2019. 2

[4] Rahaf Aljundi, Min Lin, Baptiste Goujaud, and Yoshua Bengio. Gradient based sample selection for online continual learning. In *Advances in Neural Information Processing Systems*, pages 11816–11825, 2019. 6

[5] Jihwan Bang, Heesu Kim, YoungJoon Yoo, Jung-Woo Ha, and Jonghyun Choi. Rainbow memory: Continual learning with a memory of diverse samples. In *conference on Computer Vision and Pattern Recognition*, pages 8218–8227, 2021. 1, 2

[6] Arslan Chaudhry, Marc'Aurelio Ranzato, Marcus Rohrbach, and Mohamed Elhoseiny. Efficient lifelong learning with A-GEM. In *International Conference on Learning Representations*, 2019. 1, 2

[7] 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. 2, 6, 7

[8] Danruo Deng, Guangyong Chen, Jianye Hao, Qiong Wang, and Pheng-Ann Heng. Flattening sharpness for dynamic gradient projection memory benefits continual learning. *arXiv preprint arXiv:2110.04593*, 2021. 2, 6, 7, 13

[9] Sayna Ebrahimi, Franziska Meier, Roberto Calandra, Trevor Darrell, and Marcus Rohrbach. Adversarial continual learning. In *European Conference on Computer Vision*, pages 386–402, 2020. 6

[10] Robert M French. Catastrophic forgetting in connectionist networks. *Trends in Cognitive Sciences*, pages 128–135, 1999. 1

[11] Werner H Greub. *Linear algebra*, volume 23. Springer Science & Business Media, 2012. 3

[12] Ching-Yi Hung, Cheng-Hao Tu, Cheng-En Wu, Chien-Hung Chen, Yi-Ming Chan, and Chu-Song Chen. Compacting, picking and growing for unforgetting continual learning. In *Advances in Neural Information Processing Systems*, pages 13669–13679, 2019. 1

[13] Sangwon Jung, Hongjoon Ahn, Sungmin Cha, and Taesup Moon. Continual learning with node-importance based adaptive group sparse regularization. *Advances in Neural Information Processing Systems*, 33:3647–3658, 2020. 1, 2, 6, 7

[14] Haeyong Kang, Rusty John Lloyd Mina, Sultan Rizky Hikmawan Madjid, Jaehong Yoon, Mark Hasegawa-Johnson, Sung Ju Hwang, and Chang D Yoo. Forget-free continual learning with winning subnetworks. In *International Conference on Machine Learning*, pages 10734–10750, 2022. 2

[15] 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*, pages 3521–3526, 2017. 1, 2, 6, 7

[16] Xilai Li, Yingbo Zhou, Tianfu Wu, Richard Socher, and Caiming Xiong. Learn to grow: A continual structure learning framework for overcoming catastrophic forgetting. In *International Conference on Machine Learning*, pages 3925–3934, 2019. 1, 2

[17] Guoliang Lin, Hanlu Chu, and Hanjiang Lai. Towards better plasticity-stability trade-off in incremental learning: A simple linear connector. In *conference on Computer Vision and Pattern Recognition*, pages 89–98, 2022. 1, 2, 6, 7

[18] Sen Lin, Li Yang, Deliang Fan, and Junshan Zhang. Trgp: Trust region gradient projection for continual learning. In *International Conference on Learning Representations*, 2022. 1, 2, 6, 7

[19] Hao Liu and Huaping Liu. Continual learning with recursive gradient optimization. In *International Conference on Learning Representations*, 2022. 2, 15

[20] David Lopez-Paz and Marc'Aurelio Ranzato. Gradient episodic memory for continual learning. In *Advances in Neural Information Processing Systems*, pages 6467–6476, 2017. 2, 6, 7

[21] German I Parisi, Ronald Kemker, Jose L Part, Christopher Kanan, and Stefan Wermter. Continual lifelong learning with neural networks: A review. *Neural Networks*, pages 54–71, 2019. 1, 5

[22] Jonathan D Power and Bradley L Schlaggar. Neural plasticity across the lifespan. *Wiley Interdisciplinary Reviews: Developmental Biology*, 6, 2017. 1

[23] 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. 2

[24] Gobinda Saha, Isha Garg, and Kaushik Roy. Gradient projection memory for continual learning. In *International Conference on Learning Representations*, 2021. 1, 2, 3, 4, 6, 7, 11, 12, 13

[25] Joan Serra, Didac Suris, Marius Miron, and Alexandros Karatzoglou. Overcoming catastrophic forgetting with hard attention to the task. In *International Conference on Machine Learning*, pages 4548–4557, 2018. 6, 12

[26] Pravendra Singh, Pratik Mazumder, Piyush Rai, and Vinay P. Namboodiri. Rectification-based knowledge retention for continual learning. In *conference on Computer Vision and Pattern Recognition*, pages 15282–15291, 2021. 2, 6, 7, 15

[27] Pravendra Singh, Vinay Kumar Verma, Pratik Mazumder, Lawrence Carin, and Piyush Rai. Calibrating cnns for lifelong learning. *Advances in Neural Information Processing Systems*, 33:15579–15590, 2020. 2, 6, 7, 15

[28] Tom Veniat, Ludovic Denoyer, and MarcAurelio Ranzato. Efficient continual learning with modular networks and task-driven priors. In *International Conference on Learning Representations*, 2021. 1, 2

[29] Eli Verwimp, Matthias De Lange, and Tinne Tuytelaars. Rehearsal revealed: The limits and merits of revisiting samples in continual learning. *arXiv preprint arXiv:2104.07446*, 2021. 1

[30] 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. In *Advances in Neural Information Processing Systems*, pages 22379–22391, 2021. 1, 6, 7

[31] Shipeng Wang, Xiaorong Li, Jian Sun, and Zongben Xu. Training networks in null space of feature covariance for continual learning. In *conference on Computer Vision and Pattern Recognition*, pages 184–193, 2021. 2, 3

[32] Ju Xu and Zhanxing Zhu. Reinforced continual learning. In *Advances in Neural Information Processing Systems*, pages 907–916, 2018. 1, 6, 7

[33] Jaehong Yoon, Saehoon Kim, Eunho Yang, and Sung Ju Hwang. Scalable and order-robust continual learning with additive parameter decomposition. In *International Conference on Learning Representations*, 2020. 2, 6, 7, 12

[34] Jaehong Yoon, Eunho Yang, Jeongtae Lee, and Sung Ju Hwang. Lifelong learning with dynamically expandable networks. In *International Conference on Learning Representations*, 2018. 2, 6, 7

[35] Friedemann Zenke, Ben Poole, and Surya Ganguli. Continual learning through synaptic intelligence. In *International Conference on Machine Learning*, pages 3987–3995, 2017. 1, 2

[36] Chiyuan Zhang, Samy Bengio, Moritz Hardt, Benjamin Recht, and Oriol Vinyals. Understanding deep learning (still) requires rethinking generalization. *Communications of the ACM*, pages 107–115, 2021. 3, 11

6. More Details of DualGPM and API

6.1. Relationship between Input Space and Gradient Space

According to the finding of the existing works [24, 36], the following conclusion holds:

Proposition 2. *The gradient update of the linear or convolution layer lies in the span of inputs.*

Specifically, there are two conclusions:

- The gradient update of the linear layer lies in the span of input.
- The gradient update of convolution filters lies in the space spanned by patch vectors.

Linear Layer For the linear layer, we denote its forward propagation as

$$\boldsymbol{h}_l = \sigma_l(\boldsymbol{W}_l^T \boldsymbol{h}_{l-1} + \boldsymbol{b}_l), \tag{13}$$

where σ_l denotes activation function. $\boldsymbol{W}_l \in \mathbb{R}^{d_I \times d_O}$, $\boldsymbol{h}_{l-1} \in \mathbb{R}^{d_I}$, and $\boldsymbol{h}_l \in \mathbb{R}^{d_O}$. d_I and d_O denote input and output dimension, respectively. We further denote the loss function as L. Through the chain rule, we can get the gradient of $\boldsymbol{W}_l$:

$$\begin{aligned}\frac{\partial L}{\partial \boldsymbol{W}_l} &= \frac{\partial L}{\partial \boldsymbol{h}_l}\frac{\partial \boldsymbol{h}_l}{\boldsymbol{W}_l} = \left(\frac{\partial L}{\partial \boldsymbol{h}_l} \odot \sigma_l^{'}\right)\boldsymbol{h}_{l-1}^T \\ &= \left[a_1 \boldsymbol{h}_{l-1}, a_2 \boldsymbol{h}_{l-1}, ..., a_{d_O}\boldsymbol{h}_{l-1}\right], \end{aligned} \tag{14}$$

where $\odot$ denotes element-wise multiplication. $[a_1, a_2, ..., a_{d_O}]^T$ denotes the vector $\frac{\partial L}{\partial \boldsymbol{h}_l} \odot \sigma_l^{'}$. Through (14), we can find that each column of $\frac{\partial L}{\partial \boldsymbol{W}_l}$ can be represented as input $\boldsymbol{h}_{l-1}$ multiplied by a real value a_k ($1 \le k \le d_O$). Therefore, in the linear layer, each column of the gradient $\frac{\partial L}{\partial \boldsymbol{W}_l}$ lies in the span of input. The input matrix $\boldsymbol{R}_{l,t}$ in the linear layer is got by computing $[\boldsymbol{h}_{l-1}^1, \boldsymbol{h}_{l-1}^2, ..., \boldsymbol{h}_{l-1}^N]$.

Convolution Layer For the convolution layer, we denote its forward propagation as

$$\boldsymbol{h}_l = \sigma_l(\boldsymbol{W}_l * \boldsymbol{h}_{l-1} + \boldsymbol{b}_l), \tag{15}$$

where $\boldsymbol{W}_l \in \mathbb{R}^{C_O \times C_I \times k \times k}$, $\boldsymbol{h}_{l-1} \in \mathbb{R}^{C_I \times h_I \times w_I}$, and $\boldsymbol{h}_l \in \mathbb{R}^{C_O \times h_O \times w_O}$. C_I and C_O denote input and output channels, respectively. k denotes kernel size. (h_I, w_I) and (h_O, w_O) denote the input and output size of the feature, respectively. Through the reshaping process, we can get

$$\boldsymbol{W}_l \in \mathbb{R}^{C_O \times C_I \times k \times k} \Rightarrow \hat{\boldsymbol{W}}_l \in \mathbb{R}^{(C_I \times k \times k) \times C_O}, \tag{16}$$

$$\boldsymbol{h}_{l-1} \in \mathbb{R}^{C_I \times h_I \times w_I} \Rightarrow \hat{\boldsymbol{h}}_{l-1} \in \mathbb{R}^{(C_I \times k \times k) \times (h_O \times w_O)}, \tag{17}$$

$$\boldsymbol{h}_l \in \mathbb{R}^{C_O \times h_O \times w_O} \Rightarrow \hat{\boldsymbol{h}}_l \in \mathbb{R}^{C_O \times (h_O \times w_O)}. \tag{18}$$

Then the convolution operation $*$ can be transformed into matrix multiplication, that is

$$\hat{\boldsymbol{h}}_l = \sigma_l(\hat{\boldsymbol{W}}_l^T \hat{\boldsymbol{h}}_{l-1} + \boldsymbol{b}_l). \tag{19}$$

Based on (19), the conclusion in the linear layer can be applied to the convolution layer. Since each column of $\hat{\boldsymbol{h}}_{l-1}$ is a patch vector with patch size $k \times k$ (equal to the kernel size), we can get the conclusion in the convolution layer. The input matrix $\boldsymbol{R}_{l,t}$ in the convolution layer is got by computing $[\hat{\boldsymbol{h}}_{l-1}^1, \hat{\boldsymbol{h}}_{l-1}^2, ..., \hat{\boldsymbol{h}}_{l-1}^N]$.

Please note that these two conclusions are for weight $\boldsymbol{W}_l$ and not for bias $\boldsymbol{b}_l$, so no bias units are used. Since the parameters of the neural network mainly exist in the weight of each layer, the plasticity of the neural network will be affected little without defining biases $\boldsymbol{b}_l$.

6.2. Space Transformation

We show that we can get orthogonal bases of $\mathcal{M}_{l,t}^{\perp}$ by performing SVD on the orthogonal bases of $\mathcal{M}_{l,t}$. Specifically, we can prove the following theorem.

Theorem 3. *For a subspace $\mathcal{M}$ with a set of orthogonal bases $\boldsymbol{M} = [\boldsymbol{u}_1, \boldsymbol{u}_2, ..., \boldsymbol{u}_m] \in \mathbb{R}^{d \times m}$, if we perform SVD of the matrix $\boldsymbol{M}$ ($\boldsymbol{M} = \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^T$), then the columns of $\boldsymbol{U}$ which correspond to the zero singular values form a set of orthogonal bases of $\mathcal{M}^{\perp}$. Here, $\mathcal{M}^{\perp}$ is defined as*

$$\mathcal{M}^{\perp} = \{\boldsymbol{u}^{\perp} \in \mathbb{R}^d | \, \forall \, \boldsymbol{u} \in \mathcal{M}, \ (\boldsymbol{u}^{\perp})^T \boldsymbol{u} = 0\} \tag{20}$$

Proof. Let matrix $\boldsymbol{U} = [\boldsymbol{U}_m, \boldsymbol{U}_{d-m}]$. The first m columns $\boldsymbol{U}_m$ correspond to non-zero singular values, and the last $d-m$ columns $\boldsymbol{U}_{d-m}$ correspond to zero singular values. Then the SVD of $\boldsymbol{M}$ can be rewritten as

$$\boldsymbol{M} = \left[\boldsymbol{U}_m, \boldsymbol{U}_{d-m}\right] \begin{bmatrix} \boldsymbol{\Sigma}_m \\ \boldsymbol{O} \end{bmatrix} \boldsymbol{V}^T = \boldsymbol{U}_m(\boldsymbol{\Sigma}_m \boldsymbol{V}^T), \tag{21}$$

where $\boldsymbol{O}$ represents a matrix of all zero elements.

On the one hand, since $\boldsymbol{U}$ is an orthogonal matrix, each column in $\boldsymbol{U}_m$ is orthogonal to each column of $\boldsymbol{U}_{d-m}$. Therefore, we can get $\boldsymbol{U}_m^T \boldsymbol{U}_{d-m} = \boldsymbol{O}$. According to (21), we have

$$\boldsymbol{M}^T \boldsymbol{U}_{d-m} = \boldsymbol{V}\boldsymbol{\Sigma}_m \boldsymbol{U}_m^T \boldsymbol{U}_{d-m} = \boldsymbol{O}. \tag{22}$$

Equation (22) shows that each column of $\boldsymbol{M}$ is orthogonal to each column of $\boldsymbol{U}_{d-m}$. Therefore, each column of $\boldsymbol{U}_{d-m}$ lies in the space $\mathcal{M}^{\perp}$.

On the other hand, since columns of $\boldsymbol{U}$ form a set of orthogonal bases of $\mathbb{R}^d$, any vector $\boldsymbol{v} \in \mathcal{M}^{\perp}$ can be denoted as $\boldsymbol{v} = \boldsymbol{U}\boldsymbol{a}$, where $\boldsymbol{a} \in \mathbb{R}^d$. Furthermore, due to the definition of $\mathcal{M}^{\perp}$ (see (20)), we also have

$$\boldsymbol{M}^T \boldsymbol{v} = \boldsymbol{0} \quad \Rightarrow \quad \boldsymbol{V}\boldsymbol{\Sigma}_m \boldsymbol{U}_m^T \boldsymbol{v} = \boldsymbol{0}. \tag{23}$$

Since matrix $\boldsymbol{V}\boldsymbol{\Sigma}_m$ has full column rank, we can get $\boldsymbol{U}_m^T\boldsymbol{v}=\boldsymbol{0}$ from (23). Based on these, we have

$$\boldsymbol{U}_m^T\boldsymbol{v}=\boldsymbol{0} \quad \Rightarrow \quad \boldsymbol{U}_m^T\boldsymbol{U}\boldsymbol{a}=\boldsymbol{U}_m^T\left[\boldsymbol{U}_m,\boldsymbol{U}_{d-m}\right]\begin{bmatrix}\boldsymbol{a}_m\\ \boldsymbol{a}_{d-m}\end{bmatrix} = \boldsymbol{a}_m=\boldsymbol{0}, \tag{24}$$

where $\boldsymbol{a}_m$ and $\boldsymbol{a}_{d-m}$ denote the components of $\boldsymbol{a}$ corresponding to $\boldsymbol{U}_m$ and $\boldsymbol{U}_{d-m}$, respectively. Therefore, $\boldsymbol{v}=\boldsymbol{U}_{d-m}\boldsymbol{a}_{d-m}$. Thus, any vector $\boldsymbol{v}\in\mathcal{M}^{\perp}$ is the linear combination of the columns of $\boldsymbol{U}_{d-m}$. Since we have proved that each columns of $\boldsymbol{U}_{d-m}$ lies in the subspace $\mathcal{M}^{\perp}$, columns of $\boldsymbol{U}_{d-m}$ form a set of orthogonal bases of subspace $\mathcal{M}^{\perp}$. □

6.3. Remove Subspace

In DualGPM, we need to remove subspace $\mathcal{Y}$ from $\mathcal{M}_{l,t}^{\perp}$ to get space $\mathcal{M}_{l,t+1}^{\perp}$, where $\mathcal{Y}$ denotes the subspace of $\mathcal{M}_{l,t}^{\perp}$ containing the gradient of the t-th task. We achieve this by the following theorem.

Theorem 4. *Let two spaces $\mathcal{M}$, $\mathcal{N}$ satisfy $\mathcal{N}\subseteq\mathcal{M}\subseteq\mathbb{R}^d$, where $\dim(\mathcal{M})=m$, $\dim(\mathcal{N})=n$ and $n<m\leq d$. Assume $\boldsymbol{M}=[\boldsymbol{u}_1,\boldsymbol{u}_2,...,\boldsymbol{u}_m]\in\mathbb{R}^{d\times m}$ denotes the orthogonal bases of $\mathcal{M}$, $\boldsymbol{N}=[\boldsymbol{v}_1,\boldsymbol{v}_2,...,\boldsymbol{v}_n]\in\mathbb{R}^{d\times n}$ denotes the orthogonal bases of $\mathcal{N}$. Let $\hat{\boldsymbol{M}}=\boldsymbol{M}-\boldsymbol{N}(\boldsymbol{N})^T\boldsymbol{M}$. If we perform SVD of the matrix $\hat{\boldsymbol{M}}$ ($\hat{\boldsymbol{M}}=\boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^T$), then the column vectors of $\boldsymbol{U}$ which correspond to the non-zero singular values form a set of orthogonal bases of subspace $\mathcal{O}$, where*

$$\mathcal{O}=\{\boldsymbol{u}\in\mathcal{M}|\ \forall\boldsymbol{v}\in\mathcal{N},\boldsymbol{v}^T\boldsymbol{u}=0\}. \tag{25}$$

Proof. Let matrix $\boldsymbol{U}=[\boldsymbol{U}_m,\boldsymbol{U}_{d-m}]$. The first m columns $\boldsymbol{U}_m$ correspond to non-zero singular values, and the last $d-m$ columns $\boldsymbol{U}_{d-m}$ correspond to zero singular values. Then the SVD of $\hat{\boldsymbol{M}}$ can be rewritten as

$$\hat{\boldsymbol{M}}=\left[\boldsymbol{U}_m,\boldsymbol{U}_{d-m}\right]\begin{bmatrix}\boldsymbol{\Sigma}_m\\ \boldsymbol{O}\end{bmatrix}\boldsymbol{V}^T=\boldsymbol{U}_m(\boldsymbol{\Sigma}_m\boldsymbol{V}^T), \tag{26}$$

where $\boldsymbol{O}$ represents a matrix of all zero elements.

On the one hand, it is easy to verify that

$$(\hat{\boldsymbol{M}})^T\boldsymbol{N}=\boldsymbol{M}^T\boldsymbol{N}-\boldsymbol{M}^T\boldsymbol{N}\boldsymbol{N}^T\boldsymbol{N}=\boldsymbol{0}. \tag{27}$$

With (26) and (27), we have

$$(\hat{\boldsymbol{M}})^T\boldsymbol{N}=\boldsymbol{V}\boldsymbol{\Sigma}_m^T\boldsymbol{U}_m^T\boldsymbol{N}=\boldsymbol{0}. \tag{28}$$

Since matrix $\boldsymbol{V}\boldsymbol{\Sigma}_m$ has full column rank, we can get $\boldsymbol{U}_m^T\boldsymbol{N}=0$ from (28). Therefore, each column of $\boldsymbol{U}_m$ lies in the subspace $\mathcal{O}$.

Algorithm 3 Gradient Retention Ratio

Input: Current Task $\{\mathcal{D}_t\}_{t=1}^T$, network model $f(\cdot,\boldsymbol{\Theta})$ with L layers, $\boldsymbol{\Theta}=\{\boldsymbol{W}_{l,t-1}\}_{l=1}^L$, orthogonal bases memory $\{\boldsymbol{M}_{l,t}^*\}_{l=1}^L$.
Output: Gradient Retention Ratio value $\{\mathrm{GRR}(l,t)\}_{l=1}^L$.
Initialize $\{\mathrm{GRR}(l,t)\}_{l=1}^L$ value: $\mathrm{GRR}(l,t)\leftarrow 0$;
$b\leftarrow 0$;
for $\mathcal{B}_t$ sampled from $\mathcal{D}_t$ **do**
  Compute the loss $L(\mathcal{B}_t;\boldsymbol{\Theta})$ over $\mathcal{B}_t$ and get gradient $\boldsymbol{g}_t=[\boldsymbol{g}_{1,t},\boldsymbol{g}_{2,t}...,\boldsymbol{g}_{L,t}]$;
  Using $\boldsymbol{M}_{l,t}^*$ to project gradient $\boldsymbol{g}_{l,t}$ through (3) or (4) and get $\hat{\boldsymbol{g}}_{l,t}$;
  $\mathrm{GRR}(l,t)\leftarrow\mathrm{GRR}(l,t)+\frac{||\hat{\boldsymbol{g}}_{l,t}||_2^2}{||\boldsymbol{g}_{l,t}||_2^2}$;
  $b\leftarrow b+1$;
end for
$\mathrm{GRR}(l,t)\leftarrow\mathrm{GRR}(l,t)/b$;

On the other hand, for any vector $\boldsymbol{u}\in\mathcal{O}\subseteq\mathcal{M}$, there exists a set of coefficients $\boldsymbol{a}\in\mathbb{R}^m$ such that $\boldsymbol{u}=\boldsymbol{M}\boldsymbol{a}$. Therefore, we have

$$\boldsymbol{u}=\boldsymbol{M}\boldsymbol{a}=\hat{\boldsymbol{M}}\boldsymbol{a}+\boldsymbol{N}(\boldsymbol{N})^T\boldsymbol{M}\boldsymbol{a}=\left[\boldsymbol{U}_m,\boldsymbol{N}\right]\begin{bmatrix}\boldsymbol{\Sigma}_m\boldsymbol{V}^T\boldsymbol{a}\\ (\boldsymbol{N})^T\boldsymbol{M}\boldsymbol{a}\end{bmatrix}. \tag{29}$$

This means $\boldsymbol{u}$ can be denoted as a linear combination of the columns of matrix $[\boldsymbol{U}_m,\boldsymbol{N}]$. Since $\boldsymbol{u}$ is orthogonal to the columns of $\boldsymbol{N}$ (see Definition (25)), $\boldsymbol{N}^T\boldsymbol{M}\boldsymbol{a}$ must be zero. Therefore, any vector $\boldsymbol{u}\in\mathcal{O}$ can be denoted as a linear combination of the columns of matrix $\boldsymbol{U}_m$. We have proved that each column of $\boldsymbol{U}_m$ lies in $\mathcal{O}$. Therefore, columns of $\boldsymbol{U}_m$ form a set of orthogonal bases of subspace $\mathcal{O}$. □

6.4. Algorithm for Computing Gradient Retention Ratio

We give the process of computing Gradient Retention Ratio (GRR) in Algorithm 3. Please note that this process requires only one more epoch computation.

7. More Details of Experimental Setup

7.1. Architecture Details

LeNet like Architecture LeNet-5 is used in existing work GPM [24] and ADP [33]. Besides the output layer, this network also consists of 2 convolution layers and 2 linear layers. The number of filters of the convolution layers from bottom to up is 10, 20. Both two layers have kernel size 5×5. The number of units of two linear layers is 800 and 500, respectively. After each convolution layer, 3×3 max-pooling with stride size 2 is applied.

AlexNet like Architecture This architecture is the same as several existing works [24, 25]. Specifically, the network

Table 5. Statistics of three datasets

	Split CIFAR100	CIFAR100-sup	Split Mini-Imagenet
Task Number	20	20	20
Input Size	$3 \times 32 \times 32$	$3 \times 32 \times 32$	$3 \times 84 \times 84$
Classes per Task	5	5	5
Training Samples	2375	2375	2375
Valid Samples	125	125	125
Testing Samples	500	500	500

Table 6. Statistic of 5-Datasets, the input size is set as $3 \times 32 \times 32$.

	CIFAR10	MNIST	SVHN	Fashion MNIST	notMNIST
Classes	10	10	10	10	10
Number of Samples (train)	4750	57000	69595	57000	16011
Number of Samples (val)	2500	3000	3662	3000	842
Number of Samples (test)	10000	10000	26032	10000	1873

consists of 3 convolution layers plus 2 linear layers. The number of filters of the convolution layers from bottom to up is 64, 128, and 256 with kernel sizes 4×4, 3×3, and 2×2, respectively. The number of units of two linear layers is 2048. Rectified function is used as activations for all the layers except the classifier layer. After each convolution layer, 2×2 max-pooling is applied. Dropout with ratio 0.2 is used for the first two layers and 0.5 for the rest layers.

Reduced ResNet18 Architecture This architecture is the same as existing work [24] where the parameters of ResNet18 are reduced. Specifically, each layer of reduced ResNet18 is three times fewer features than the original architecture. The average-pooling before the classifier layer is set as 2×2.

7.2. Datasets Statistic

We give detailed statistics of datasets in this section. Specifically, Table 5 shows the detailed information of three datasets, including CIFAR100-sup, Split CIFAR100, and Split Mini-Imagenet. These datasets are constructed by one dataset (CIFAR100 or Mini-Imagenet). Table 6 shows the detailed information of 5-Datasets. This dataset is constructed by five different datasets and each dataset forms a task of 5-Datasets.

7.3. Threshold Setting for Orthogonal Bases Updating

The set of threshold ϵ_{th} follows existing work GPM [24]. For the experiment of Split CIFAR100 with AlexNet architecture, threshold ϵ_{th} is set as 0.97 for all the layers and increased by 0.0015 for each new task. For the experiment of CIFAR100-sup with LeNet, ϵ_{th} is set as 0.98 for the first layer and increased by 0.001 for each new task. For the experiment of Split Mini-Imagenet with ResNet18, ϵ_{th} is set as 0.985 for the first layer and increased by 0.0003 for each new task. For the experiment of 5-Dataset with ResNet18 architecture, threshold ϵ_{th} is set as 0.965 for all the layers.

7.4. Hyper-Parameters

We give the hyper-parameters for each method in Table 7. SCIFAR100 denotes Split CIFAR100 and mini denotes Split Mini-Imagenet. FS-DGPM, GPM and TRGP are implemented by their official codes under MIT License. EWC, GEM, ER-Res are implemented by the code provided by FS-DGPM [8] under MIT License.

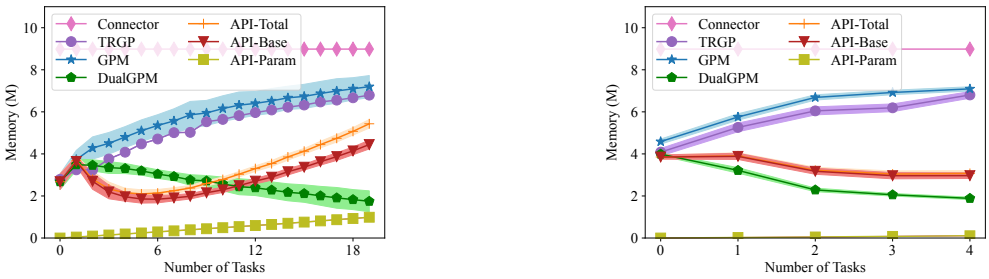
8. Additional Experimental Results

8.1. Memory Usage

Variation of Memory We show the variation of memory usage in the experiment of Split Mini-Imagenet and 5-Datasets. The memory usage of GPM increases all the time. The memory usage of DualGPM increases first and decreases later. However, the memory usage of API on Split Mini-Imagenet differs from that of API on Split CIFAR100 and 5-Datasets. Specifically, API-Base first increases, then decreases, and finally increases again. The second increase in API-Base is because the dimension of bases increases with the expansion of parameters $\boldsymbol{W}_{l,t}$ (see Equation (9)).
Variation of Bases We give the variation of bases in different layers. Specifically, we show the variation of bases in

Table 7. List of hyper-parameters for different methods

Methods	Hyper-Parameters
EWC	lr: 0.03 (5-Datasets), 0.05 (SCIFAR100, CIFAR100-sup), 0.1 (mini) regular: 300 (SCIFAR100, CIFAR100-sup), 2000 (mini), 5000 (5-Datasets)
ER-Res	lr: 0.05 (SCIFAR100, CIFAR100-sup), 0.1 (5-Datasets, mini) memory: 2000 (SCIFAR100, CIFAR100-sup, mini), 3000 (5-Datasets)
GPM	lr: 0.01 (SCIFAR100, CIFAR100-sup), 0.1 (5-Datasets, mini)
GEM	lr: 0.05 (SCIFAR100, CIFAR100-sup) memory strength: 0.5 (SCIFAR100, CIFAR100-sup) memory: 2000 (SCIFAR100, CIFAR100-sup)
A-GEM	lr: 0.05 (SCIFAR100, CIFAR100-sup), 0.1 (5-Datasets, mini) memory: 2000 (SCIFAR100, CIFAR100-sup, mini), 3000 (5-Datasets)
FS-DGPM	lr, η_3: 0.01 (SCIFAR100, CIFAR100-sup) lr for sharpness, η_1: 0.001 (SCIFAR100), 0.01 (CIFAR100-sup) lr for DGPM, η_2: 0.01 (SCIFAR100, CIFAR100-sup) memory: 900 (SCIFAR100), 1100 (CIFAR100-sup)
TRGP	lr: 0.01 (SCIFAR100, CIFAR100-sup), 0.1 (5-Datasets, mini) K: 2 (SCIFAR100, CIFAR100-sup, 5-Datasets, mini) ϵ: 0.5 (SCIFAR100, CIFAR100-sup, 5-Datasets, mini)
API	lr: 0.01 (SCIFAR100, CIFAR100-sup), 0.1 (5-Datasets, mini) K: 10 (SCIFAR100, CIFAR100-sup, 5-Datasets, mini) ρ: 0.5 (SCIFAR100, CIFAR100-sup, 5-Datasets, mini)



(a) Variation of memory usage

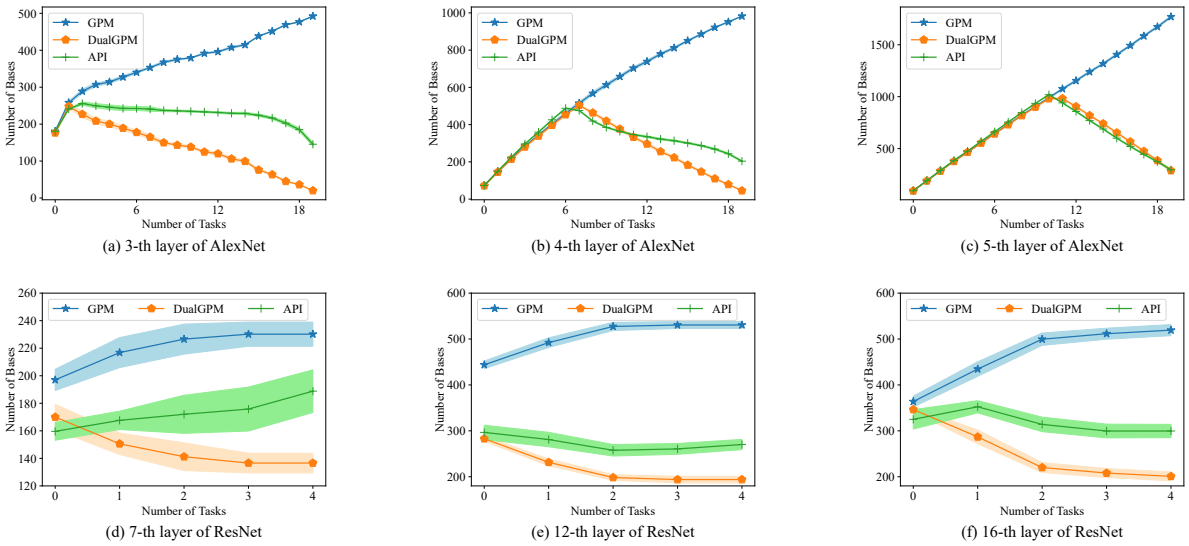
(b) Variation of memory usage

Figure 7. (a) Variation of memory usage for Split Mini-Imagenet. (b) Variation of memory usage for 5-Dataset.

the 3-rd, 4-th, and 5-th layers of AlexNet when the model learns on Split CIFAR100. We also show the variation of bases in the 7-th, 12-th, and 16-th layers of ResNet when the model learns on 5-Datasets. We can find that GPM increases the bases all the time, while DualGPM and API do not increase bases all the time and keep much fewer bases than GPM in these layers.

8.2. Time Consumption

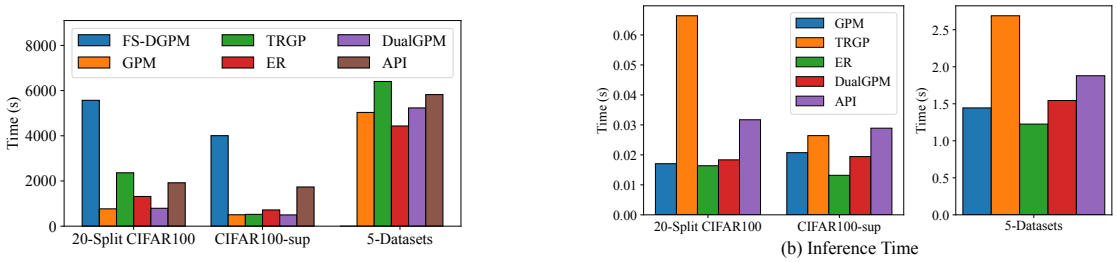
API expands the model's parameters in each layer, and the new task has access to more parameters to learn the model than the old task. However, this increases the time consumption in both the training and inference phases. Figure 9 gives the time consumption for different methods. We can find that API does consume more time than GPM in both the training and inference phases. However, the time consumption of API is comparable with other methods. Specifically, the training time of API is much less than FS-DGPM on Split CIFAR100 and CIFAR100-sup. The training time of API is slightly less than that of TRGP on Split CIFAR100 and 5-Datasets, and slightly larger than that of TRGP on CIFAR100-sup. The average inference time of API is less than TRGP on Split-CIFAR100 and 5-Datasets, and slightly larger than TRGP on CIFAR100-sup.



(a) 3-th layer of AlexNet (b) 4-th layer of AlexNet (c) 5-th layer of AlexNet

(d) 7-th layer of ResNet (e) 12-th layer of ResNet (f) 16-th layer of ResNet

Figure 8. Variation of bases in different layers of AlexNet and ResNet.



(a) Training Time (b) Inference Time

Figure 9. (a) Training time of different methods on three different datasets. (b) Average inference time of different methods on three different datasets.

8.3. More Comparison with Expansion-Based Methods

We choose calibrating CNNs for lifelong learning (CCLL) [27] and rectification-based knowledge retention (RKR) [26] to compare with our API. Other expansion-based methods have different experimental settings from ours, and many hyperparameters are difficult to tune. Besides, some methods that define a search space and use some methods to search for expansion strategies incur high computational costs. On the contrary, CCLL and RKR are simple and easy to implement, and they have demonstrated that they outperform many expansion-based methods.

We compare API with CCLL and RKR on Split CIFAR100 and Split Mini-Imagenet. The experimental setting is the same as that of the experiments in the paper. We tune the hyperparameter K in these two methods so that their model capacity is not smaller than that of our method. Please note that larger K in these two methods means larger model capacity and higher accuracy. The results are given in Table 8. We can find that API also gets the best results with the smallest model capacity.

8.4. More Comparison with Algorithm-Based Methods

Recursive gradient optimization (RGO) [19] is also an algorithm-based method that rectifies new task gradient layer by layer to overcome catastrophic forgetting. This method directly maintains projection matrix $\boldsymbol{P}_l$ in memory. Please note that GPM maintains orthogonal bases $\boldsymbol{M}_l$ in memory and DualGPM maintains orthogonal bases $\boldsymbol{M}_l$ or $\boldsymbol{M}_l^{\perp}$ in memory. When learning the new task, GPM and DualGPM get projection matrix $\boldsymbol{P}_l = \boldsymbol{M}_l \boldsymbol{M}_l^T$.

Compared with RGO, GPM and DualGPM use less memory. Specifically, RGO keeps $\boldsymbol{P}_l$ in memory and requires d_l^2 parameters. GPM keeps $\boldsymbol{M}_l$ in memory and requires $\dim(\mathcal{M}_l)d_l$ parameters. DualGPM keeps $\boldsymbol{M}_l$ or $\boldsymbol{M}_l^{\perp}$ in memory and requires $\min(\dim(\mathcal{M}_l), \dim(\mathcal{M}_l^{\perp}))d_l$

Table 8. The performance for different expansion-based methods on Split CIFAR100 dataset and Split Mini-Imagenet.

	SPLIT CIFAR100		SPLIT MINI-IMAGENET	
METHODS	ACC (%)	CAPACITY (%)	ACC (%)	CAPACITY (%)
RKR	77.5 ± 0.5	107	63.7 ± 2.0	132
CCLL	75.4 ± 0.3	105	65.7 ± 0.5	132
API	$\mathbf{81.4 \pm 0.4}$	**104**	$\mathbf{65.9 \pm 0.6}$	**127**

Table 9. The performance for different algorithm-based methods on Split CIFAR100 dataset and Split Mini-Imagenet.

	SPLIT CIFAR100		SPLIT MINI-IMAGENET	
METHODS	ACC (%)	AVG-MEMORY (M)	ACC (%)	AVG-MEMORY (M)
RGO	$\mathbf{81.4 \pm 0.2}$	6.3	58.3 ± 2.5	9.0
GPM	78.9 ± 0.2	2.9	61.2 ± 0.6	5.7
DUALGPM	78.5 ± 0.4	**1.7**	61.2 ± 0.6	**2.6**
API	$\mathbf{81.4 \pm 0.4}$	2.1	$\mathbf{65.9 \pm 0.6}$	3.3

parameters in memory. Since

$$\min(\dim(\mathcal{M}_l), \dim(\mathcal{M}_l^{\perp})) \leq \dim(\mathcal{M}_l) \leq d_l, \quad (30)$$

GPM and DualGPM use less memory than RGO. Furthermore, since $\dim(\mathcal{M}_l)$ increases with the increase of tasks, DualGPM uses much less memory than RGO and GPM when the number of tasks is large.

We compare RGO with our methods on Split CIFAR100 and Split Mini-Imagenet. We use the official implementation of RGO and keep its architecture consistent with our methods. Table 9 gives the comparison between RGO and our methods. 'AVG-MEMORY' denotes the average memory used when the model learns each new task. RGO gets similar accuracy to API on Split CIFAR100 but uses much more memory than GPM and our method. On Split Mini-Imagenet, RGO gets lower accuracy than our methods and GPM, and its average memory usage is more than GPM, DualGPM and API.

8.5. More Plasticity Evaluation Results

In Section 3.2 of the paper, we present the relationship between the model's performance and AGRR when the model is trained on task 2 of Split-CIFAR100 with DualGPM (non-expandable parameters). Here, we give the details and more results of this experiment.

All experimental settings were the same as that in Section 4, except that ϵ_{th}^l (see 6 and 8) is adjusted to 6 different values, including 0.97, 0.975, 0.98, 0.985, 0.99, 0.995. The experiment is performed three times with each value of ϵ_{th}^l. Therefore, there are 18 points in Figure 3. Average gradient norm denotes the average of the norm of the gradient

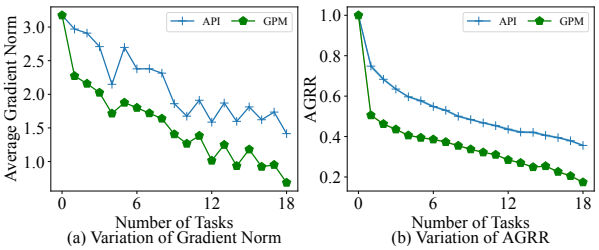


Figure 10. (a) Variation of average gradient norm on Split CIFAR100. (b) Variation of AGRR on Split CIFAR100.

used for updating parameters during the learning of a task. Specifically, we use $\frac{1}{S}\sum_{i=1}^{S}||\hat{\boldsymbol{g}}_i||_2$ to denote the average gradient norm, where S denotes the update times. Obviously, the larger the average gradient norm is, the larger the model updates the parameters.

In Figure 11, we give the results on more tasks. We can find that the conclusion is consistent. Specifically, when the constraint increases (AGRR decreases), the model becomes more conservative in updating parameters, and the accuracy of the model on this task also shows a downward trend.

8.6. Variation of AGRR

We show the variation of AGRR and average gradient norm during the learning of Split CIFAR100 in Figure 10. We can find that both AGRR and average gradient norm decrease with the increase of tasks. However, our method API adaptively improves the model's plasticity. Therefore, AGRR and average gradient norm are larger than GPM during the whole learning process.

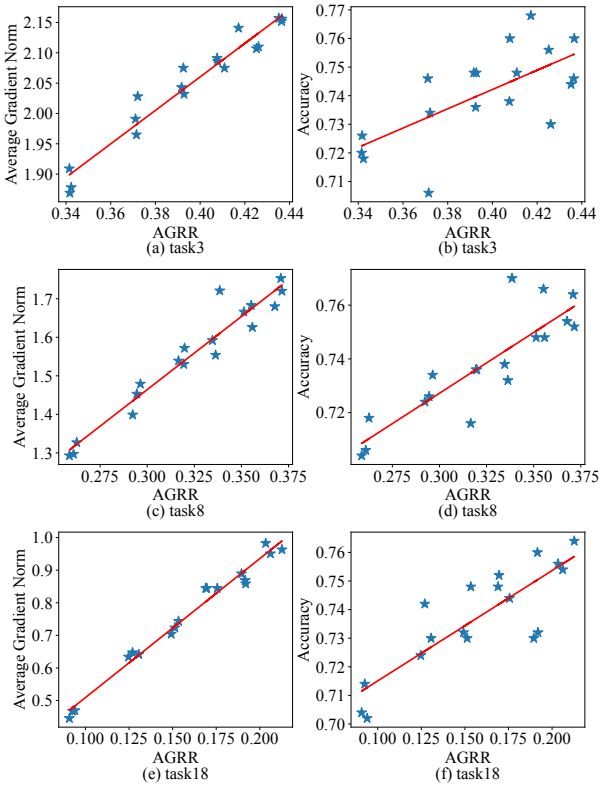


Figure 11. DualGPM with non-expandable parameters learns on Split CIFAR100. (a), (c) and (e) show the correlation between AGRR and average gradient norm for learning different tasks. (b), (c) and (d) show the correlation between AGRR and the accuracy of different tasks.