Memory-efficient Continual Learning with Prototypical Exemplar Condensation

Supplementary Material

A. Related Works

A.1. Continual Learning

Continual Learning (CL) aims to enable a neural network model to learn new tasks continuously without forgetting the old task. Inspired by the working mechanism of the human brain, mainstream memory-based CL methods consolidate previously learned knowledge by replaying old data, thereby avoiding catastrophic forgetting. Due to strict memory and privacy constraints, usually only a small portion of old task data can be kept, many above-mentioned coreset selection methods are used to select informative samples for storage. For example, some heuristic method is used to select the most representative samples from each classes or the sample closest to the decision boundary.

Experience Replay. According to the representative sampling, BCSR [22] proposes a bilevel optimization framework for coreset selection, where coreset construction is formulated as an upper-level selection problem with model training at the lower level. OCS [54] leverages gradient-based criteria (i.e., minibatch similarity, sample diversity, and coreset affinity) to select a high-quality coreset at each iteration, making the model more robust to noisy and imbalanced data streams. PBCS [59] CSReL [40] quantifies the potential performance gain a sample can provide, allowing the method to select high-value samples while avoiding ambiguous or noisy outliers which often degrade performance in traditional bi-level optimization approaches. According to the boundary-based sampling, RM [2] propose a diversity-aware sampling method for effectively managing the memory with limited capacity by leveraging classification uncertainty. However, the number of new task's samples and stored old samples in memory is often highly unbalanced, leading the model to be biased towards learning new tasks with more data.

LPR [53] employs a replay-tailored preconditioner applied to the loss gradients. This preconditioner is designed to balance two key objectives: effectively learning from both new and past data, while ensuring that predictions (and internal representations) of past data are modified only gradually, thereby enhancing optimization stability and efficiency. DEMD [51] proposes an architecture that integrates a dynamic memory system for retaining the dynamic information alongside a PuriDivER [3] introduces a method for preserving a set of training examples that are both diverse and label-pure. This is achieved through a scoring function that primarily promotes label purity, while incorporating an additional term to encourage diversity by optimizing the sample distribution to resemble that of noisy examples. RAR [55] replays stored samples with data augmentation to enhance memory utilization.

Generative Replay. Generative replay usually requires training an additional generative model to replay generated data. This is closely related to CL of generative models themselves as they also require incremental updates. DGR [37] uses a generative model to synthesize realistic past data. Instead of storing actual old samples, the model is jointly trained on new data and the pseudo-samples generated on-demand. MeRGAN [47] employs replay alignment to enforce consistency of the generative data sampled with the same random noise between the old and new generative models, similar to the role of function regularization.

Gradient Replay. GEM [28], LOGD [39], MER [35] constrain parameter updates to align with the direction of experience replay, corresponding to preserving the previous input space and gradient space with some old training samples. GPM [36] maintains the gradient subspace important to the old tasks (i.e., the bases of core gradient space) for orthogonal projection in updating parameters. VR-MCL [48] introduces an approach that implicitly enhances the online Hessian approximation in continual learning by applying a variance reduction technique. This method, which refines the efficiency and stability of weight updates, is designed to mitigate catastrophic forgetting more effectively in non-stationary online data streams.

A.2. Dataset Distillation

Dataset distillation or condensation aims to condense a large dataset into much smaller yet informative one. It finds applications in neural architecture search, continual learning and privacy protection, etc.

SRe2L [52] introduces a three-stage paradigm that leverages highly encoded distribution priors to bypass the need for supervision typically provided during model training. NRR-DD [41] adopts a distance-based strategy to quantify the discrepancy between predictions on synthetic data and one-hot encoded labels, thereby streamlining the training process and minimizing label storage requirements. Zhu et al. [61] observe that dataset distillation often discards semantically meaningful information, and consequently propose masking techniques that enhance calibration with minimal computational overhead. IGD [10] is designed to guide diffusion models directly during data distillation under a generalized training-effective condition, eliminating

the need for model retraining. Qin et al. [33] propose a data-knowledge scaling law to quantify the extent to which external knowledge can reduce dataset size. CUDD [30] introduces a strategic curriculum-based approach for distilling synthetic images, progressing from simple to complex samples. Gu et al. [20] present a dataset distillation framework for diffusion models based on an auxiliary minimax criterion, aimed at enhancing the representativeness and diversity of generated data. Du et al. [16] propose a dynamic adjustment mechanism that improves the diversity of synthesized datasets with negligible computational cost, leading to notable performance gains. H-PD [57] systematically explores hierarchical features within the latent space and optimizes them in GANs using loss signals derived from the dataset distillation task. Finally, EDF [43] builds on trajectory matching principles by prioritizing updates to discriminative regions, guided by gradient weights extracted from Grad-CAM activation maps. Deng et al. [14] introduce the class centralization constraint, which encourages the clustering of samples belonging to the same class in order to enhance class discrimination. To address the limitations of existing methods in aligning feature distributions, they further propose a covariance matching constraint, which facilitates more accurate alignment between real and synthetic datasets by leveraging local feature covariance matrices.

B. Rethinking of memory in continual learning

B.1. Scaling of Memory in Continual Learning

A central challenge in CL lies in the scalability of memory usage as the number of tasks grows. One of the standard approaches for managing the replay buffer is reservoir sampling [6, 7], where incoming data are inserted into the buffer with a uniform probability, and existing exemplars are randomly discarded once the memory is full. However, this method has been shown to suffer from a noticeable decline in performance, particularly when the number of tasks is relatively small (fewer than 20) [6]. This degradation mainly arises from the random nature of exemplar replacement, which can lead to the premature removal of informative samples and reduce the representational quality of the stored buffer.

Alternatively, coreset selection strategies allocate a fixed number of exemplars per class [2, 40]. This approach generally achieves significantly better performance than reservoir sampling, since the memory is constructed from samples that are more salient to the learning process. However, these methods typically incur linear growth in memory requirements with respect to the number of tasks, i.e., $\mathcal{O}(CN_C)$, where C denotes the number of classes and N_C the number of stored samples per class. In practice, coreset selection usually requires up to 100 exemplars per class to maintain competitive performance [2]. When the number of exemplars per class is reduced to around 20, however, the model suffers a substantial accuracy drop by 20% according to empirical observations [5]. This strategy, while effective in mitigating catastrophic forgetting, becomes impractical in real-world deployments where tasks and number of classes C accumulate over time and storage is constrained, as is common in edge devices and mobile platforms.

To address this limitation, we shift our focus toward prototype-based memory, which aims to minimize the number of stored samples per class while preserving the representational capacity needed for effective continual learning.

B.2. Prototypes in Continual Learning

Fundamentally, prototype-based memory [1, 9, 45] has gained the big interest to further enhance memory efficiency thanks to several advantages:

1. prototypes offer a lightweight alternative. Instead of storing entire data samples, we can reduce the memory footprint significantly [9].
2. Since the prototype captures the central tendency of the data within each class distribution, it can naturally facilitate data augmentation through simple transformation techniques, such as Gaussian noise injection [62]. Consequently, synthetic subsets can be generated by perturbing samples around the prototype, effectively producing diverse data points from a single exemplar.
3. Prototypes provide a stable reference point for previously learned knowledge. By maintaining these prototypes, the model can preserve the structure of the embedding space. When learning a new task, the model is encouraged to keep new class embeddings close to their own prototypes while maintaining a clear distance from the prototypes of old classes. This approach not only minimizes storage requirements but also maintains high performance in continual learning scenarios.

Although prototypes have demonstrated robustness in CL, their use remains underexplored in current research, primarily due to several limiting factors:

1. **Prototypes as guidance for rehearsal replay.** When prototypes are employed in rehearsal-based replay, they primarily serve two purposes. The first is to guide the optimization of the feature extractor [24, 27]. However, this approach does not fully replace traditional rehearsal methods that store raw samples, as the guidance process still relies on previous samples to compute the necessary updates.
2. **Prototypes as stored exemplars.** When prototypes are directly used as stored exemplars for replay, the backpropagation process is applied only to the classifier layer [9, 23]. Consequently, the feature extractor cannot be effectively updated, which limits the model's capacity to adapt and retain knowledge across tasks.
3. **Vulnerability to catastrophic forgetting.** The con-

struction of prototypes depends on effectively trained prototypical networks. However, similar to other parametric models, prototypical networks are susceptible to catastrophic forgetting [12, 45]. As new tasks are introduced, previously learned prototypes may be overwritten or drift within the representation space, leading to a substantial performance decline on earlier tasks. This problem becomes increasingly severe as the number of classes grows or the task sequence lengthens.

C. Notations

Table 6. Notations used in the proposed algorithms.

Notation	Description
T	Total number of tasks.
$\lvert\mathcal{T}_t\rvert$	Number of samples per task t.
$\mathcal{T}_t$	Dataset of task t with $\lvert\mathcal{T}_t\rvert$ samples.
$\mathcal{C}_t$	Set of available class for task t.
$\mathcal{C}$	Set of available class.
y_i^t	Class label of sample x_i^t.
$f_\theta(\cdot)$	Feature extractor with parameters θ.
$f_\phi(\cdot)$	Classifier with parameters ϕ.
$g(\cdot)$	Pretrained image decoder.
$\mathcal{M}$	Memory buffer storing real exemplars $\mathcal{M}_x$ and synthetic exemplars $\mathcal{M}_s$.
$\mathcal{M}_x, \mathcal{M}_x^c$	Memory of all real exemplars, and real exemplars according to class c.
$\mathcal{M}_s$	Memory of synthetic exemplars.
$\mathcal{M}_p$	Memory of real prototypes.
$\mathcal{B}_m$	Batch sampled from memory at previous task m.
s^c	Currently trained synthetic exemplar for class c.
$\hat{s}^c$	Stored synthetic exemplar for class c.
p^c	Prototype of class c computed from real exemplars of the current class t.
$\hat{p}^c$	Prototype of class c computed from synthetic exemplars in M_s.
η	Learning rate.
E	Number of training epochs per task.

D. Algorithm Details

We present the detailed procedure of ProtoCore in Algorithm 4. We present the detailed procedure of ProtoCore + replayed-based CL in Algorithm 1

D.1. Full Memory Replay

Algorithm 1: ProtoCore (Replay memory supported). The blue annotations reveals the model update of continual learning. The orange annotations demonstrates the prototypical exemplar generation.

Input: Feature extractor f_θ, classifier f_ϕ, pretrained image encoder g, number of tasks T, training set $\mathcal{T} = \{\{(x_i^t, y_i^t)\}_{i=1}^{|\mathcal{T}_t|}\}_{t=1}^{T}$, $y_i^t \in \mathcal{Y}$, $\mathcal{M} = \mathcal{M}_s \bigcup \mathcal{M}_x \bigcup \mathcal{M}_p$.

Output: θ^T;

// iterative approximation

for t *in* $1, ..., T$ **do**
- Obtain past model $\theta' \leftarrow \theta^{t-1}$.
- Obtain current task data $\mathcal{T}_t = \{(x_i^t, y_i^t)\}_{i=1}^{|\mathcal{T}_t|}$.
- Obtain previous task data $\{\{(\hat{x}_i^t, \hat{y}_i^t)\}_{i=1}^{|\mathcal{B}_m|}\}_{m=1}^{t-1} \leftarrow \mathcal{M}_x$.
- // Init learnable data at begin of every task
- Initialize learnable synthetic latent z, optimizer $\texttt{opt}(z)$, image decoder $s = g(z)$.
- **for** e *in* $0, \ldots, E-1$ **do**
  - **for** *batch in* $\mathcal{T}_t$ **do**
    - Obtain (x, y) from batch.
    - // Loss Computation
    - Compute $\mathcal{L}_{\texttt{total}}$ according to Alg. 2.
    - // Model update
    - $\theta' = \theta' - \eta \nabla_{\theta'} \mathcal{L}_{\texttt{total}}$
- $\theta^t \leftarrow \theta'$.
- **for** e *in* $0, \ldots, E-1$ **do**
  - **for** *sample in* $\mathcal{T}_t$ **do**
    - Obtain (x, y) from sample.
    - // Prototypical Exemplar Loss Computation
    - Compute $\mathcal{L}_{\texttt{sync\_proto}}$ according to Alg. 3.
    - // Optimize Prototypical Exemplar
    - $s = s - \eta \nabla_s \mathcal{L}_{\texttt{sync\_proto}}$
- // Store exemplars and data.
- $\mathcal{M}_x \leftarrow \texttt{reservoir}(\mathcal{M}, (x, y) \in \mathcal{T}_t)$.
- $\mathcal{M}_s \leftarrow \texttt{replace}(\mathcal{M}, (s, y))$.
- $\mathcal{M}_p \leftarrow \texttt{replace}(\mathcal{M}, (p))$.

Algorithm 2: Network update with prototypical exemplar alignment (Replay memory supported). The blue annotations demonstrates the task-head learning. The orange annotations refers to the prototype learning for the feature extractor.

Input: Feature extractor f_θ, classifier f_ϕ, current task $\mathcal{T}_t$, $y_i^t \in \mathcal{Y}$, $\mathcal{M} = \mathcal{M}_s \bigcup \mathcal{M}_x \bigcup \mathcal{M}_p$. Coefficients $\alpha_1, \alpha_2, \alpha_3, \beta_1, \beta_2$.

Output: $\mathcal{L}_{\texttt{total}}$;

// initialization

for c *in* $\mathcal{C}^t$ *seen classes* **do**
- Samples $\{\hat{x}_i^c\}_{i=1}^{|\mathcal{M}_x^c|}$ according to class c in $\mathcal{M}_x$.
- Samples $\hat{s}^c$ according to classes c in $\mathcal{M}_s$.

// Task-head Learning with Memory

// Current Task

$\mathcal{L}_{\texttt{task\_cur}} = 0$.

for $(x; y)$ *in* $\mathcal{T}_t$ **do**
- $\mathcal{L}_{\texttt{task\_cur}} \mathrel{+}= \texttt{CE}(f_\phi(f_\theta(x)), y)$.

// Previous Tasks

$\mathbf{z} = \{f_\theta(h(s)) | \forall h \in \mathcal{F}\}$.

$\mathcal{L}_{\texttt{task\_pre}} \mathrel{+}= \texttt{CE}(f_\phi(\mathbf{z}), y)$.

// Full Memory Replay

for $(x; y)$ *in* $\mathcal{M}_x$ **do**
- $\mathcal{L}_{\texttt{task\_pre}} \mathrel{+}= \texttt{CE}(f_\phi(f_\theta(x)), y)$.

// Incremental Feature Extractor

// Assign prototypes to list

for c *in* $\mathcal{C}^{1:t-1}$ *seen classes* **do**
- Samples $\hat{s}^c$ according to classes c in $\mathcal{M}_s$.
- **if** c *is in* $\mathcal{C}^t$ *current classes* **then**
  - Samples $\mathcal{T}_t^c$ according to classes c in $\mathcal{T}_t$.
  - $p^c = \frac{\beta_1}{|\mathcal{T}_t^c|} \sum_{(x;\cdot) \sim \mathcal{T}_t^c} f_\theta(x) + \frac{\beta_2}{|\mathcal{F}|} \sum_{h \sim \mathcal{F}} f_\theta(h(\hat{s}^c))$.
- **else**
  - $p^c = \frac{\beta_2}{|\mathcal{F}|} \sum_{h \sim \mathcal{F}} f_\theta(h(\hat{s}^c))$.

// Prototypical Loss

$\mathcal{L}_{\texttt{cur\_pro}}, \mathcal{L}_{\texttt{pre\_pro}} = 0, 0$.

for c *in* $\mathcal{C}^t$ *seen classes* **do**
- **if** c *is in* $\mathcal{C}^t$ *current classes* **then**
  - **for** $(x; y)$ *in* $\mathcal{T}_t$ *according to* c **do**
    - $\mathcal{L}_{\texttt{cur\_pro}} \mathrel{+}= \frac{\exp(-\texttt{MSE}(f_\theta(x); p^c))}{\sum_{l' \neq l} \exp(-\texttt{MSE}(f_\theta(x); p^{l'}))}$.
- **else**
  - **for** $(s; y)$ *in* $\mathcal{M}_s$ **do**
    - $\mathbf{z} = \{f_\theta(h(s)) | \forall h \in \mathcal{F}\}$
    - $\mathcal{L}_{\texttt{pre\_pro}} \mathrel{+}= \frac{\exp(-\texttt{MSE}(\mathbf{z}; p^c))}{\sum_{l' \neq l} \exp(-\texttt{MSE}(\mathbf{z}; p^{l'}))}$.
  - **for** $(x; y)$ *in* $\mathcal{M}_x$ *according to* c **do**
    - $\mathcal{L}_{\texttt{pre\_pro}} \mathrel{+}= \frac{\exp(-\texttt{MSE}(f_\theta(x); p^c))}{\sum_{l' \neq l} \exp(-\texttt{MSE}(f_\theta(x); p^{l'}))}$.

return $\mathcal{L}_{\texttt{total}} = \mathcal{L}_{\texttt{cur\_pro}} + \alpha_1 \mathcal{L}_{\texttt{pre\_pro}} + \alpha_2 \mathcal{L}_{\texttt{task\_pre}} + \alpha_3 \mathcal{L}_{\texttt{task\_cur}}$

Algorithm 3: Prototypical Exemplar Generation.

Input: Feature extractor f_θ, classifier f_ϕ, current task $\mathcal{T}_t$, $y_i^t \in \mathcal{Y}$, synthetic data $s = \{s^c\}_{l=1}^{\mathcal{C}^t}$ and optimizer $\texttt{opt}(s)$, coefficient α_1, α_2, $\mathcal{M} = \mathcal{M}_s \bigcup \mathcal{M}_x \bigcup \mathcal{M}_p$.

Output: θ^{t+T};

`// Current Exemplar Alignment`

for *c in current classes of* $\mathcal{T}_t$ **do**
- Samples $\mathcal{T}_t^c$ according to classes c in $\mathcal{T}_t$.
- $p^c = \frac{1}{n^c} \sum_{(x,y)\sim\mathcal{T}_t^c} f_\theta(x)$.
- $\mathcal{L}_{\texttt{cur\_syn}} \mathrel{+}= \texttt{MSE}(f_\theta(s^c); p^c)$.

`// Previous Exemplar Alignment`

for *c in* $\mathcal{C}^{1:t-1}$ *seen classes* **do**
- **if** *c is in current classes* **then**
  - Samples $\hat{s}^c$ according to classes c in $\mathcal{M}_s$.
  - $\mathcal{L}_{\texttt{pre\_syn}} \mathrel{+}= \texttt{MSE}(f_\theta(s^c); f_\theta(\hat{s}^c))$.

`// Representation Shift Alignment`

for *c in* $\mathcal{C}^{1:t-1}$ *seen classes* **do**
- **if** *c is in current classes* **then**
  - Samples $\hat{s}^c$ according to classes c in $\mathcal{M}_s$.
  - Samples $\hat{p}^c$ according to classes c in $\mathcal{M}_p$.
  - $\mathcal{L}_{\texttt{shift}} \mathrel{+}= \texttt{MSE}(f_\theta(s^c); \hat{p}^c)$.

return

$\mathcal{L}_{\texttt{sync\_proto}} = \mathcal{L}_{\texttt{cur\_syn}} + \alpha_1 \mathcal{L}_{\texttt{pre\_syn}} + \alpha_2 \mathcal{L}_{\texttt{shift}}$

D.2. Prototypical-Exemplar Only Replay

Algorithm 4: ProtoCore (Synthetic Replay Only). The blue annotations reveals the model update of continual learning. The orange annotations demonstrates the prototypical exemplar generation.

Input: Feature extractor f_θ, classifier f_ϕ, number of tasks T, data $\mathcal{T} = \{\{(x_i^t, y_i^t)\}_{i=1}^{N_t}\}_{t=1}^{T}$, $y_i^t \in \mathcal{Y}$, memory pool $\mathcal{M} = \mathcal{M}_s$.
Output: θ^T;

// initialization
$f_\theta, f_\phi, \mathcal{M} = \emptyset$
// iterative approximation
for t *in* $1, ..., T$ **do**
  Obtain past model $\theta' \leftarrow \theta^{t-1}$.
  Obtain current task data $\mathcal{T}_t = \{(x_i^t, y_i^t)\}_{i=1}^{n_t}$.
  // Init learnable data at begin of every task
  Initialize learnable synthetic data s, optimizer $\texttt{opt}(s)$.
  for e *in* $0, \ldots, E-1$ **do**
    for *sample in* $\mathcal{T}_t$ **do**
      Obtain (x, y) from sample.
      // Loss Computation
      Compute $\mathcal{L}_{\texttt{total}}$ according to Alg. 2.
      // Model update
      $\theta' = \theta' - \eta \nabla_{\theta'} \mathcal{L}_{\texttt{total}}$
  $\theta^t \leftarrow \theta'$.
  for e *in* $0, \ldots, E-1$ **do**
    for *sample in* $\mathcal{T}_t$ **do**
      Obtain (x, y) from sample.
      // Prototypical Exemplar Loss Computation
      Compute $\mathcal{L}_{\texttt{sync\_proto}}$ according to Alg. 3.
      // Optimize Prototypical Exemplar
      $s = s - \eta \nabla_s \mathcal{L}_{\texttt{sync\_proto}}$
  // Store exemplars and data.
  $\mathcal{M}_s \leftarrow replace(\mathcal{M}, (s, y))$.

Algorithm 5: Network update with prototypical exemplar alignment (Synthetic Replay Only). The blue annotations demonstrates the task-head learning. The orange annotations refers to the prototype learning for the feature extractor.

Input: Feature extractor f_θ, classifier f_ϕ, current task $\mathcal{T}_t$, $y_i^t \in \mathcal{Y}$, memory $\mathcal{M} = \mathcal{M}_s$. Coefficients $\alpha_1, \alpha_2, \alpha_3, \beta$, Transformation set $\mathcal{F}$.
Output: $\mathcal{L}_{\texttt{total}}$;

// initialization
for c *in* $\mathcal{C}^t$ *seen classes* **do**
  Samples $\hat{s}^c$ according to classes c in $\mathcal{M}_s$.
// Task-head Learning with Memory
// Current Task
$\mathcal{L}_{\texttt{task\_cur}} = 0$.
for $(x; y)$ *in* $\mathcal{T}_t$ **do**
  $z = f_\theta(x)$.
  $\mathcal{L}_{\texttt{task\_cur}} \mathrel{+}= \mathcal{L}_{CE}(f_\phi(z), y)$.
// Previous Tasks
$\mathcal{L}_{\texttt{task\_pre}} = 0$.
$\mathbf{z} = \{f_\theta(h(s)) | \forall h \in \mathcal{F}\}$.
$\mathcal{L}_{\texttt{task\_pre}} \mathrel{+}= \texttt{CE}(f_\phi(\mathbf{z}), y)$.
// Incremental Feature Extractor
// Assign prototypes to list
for c *in* $\mathcal{C}^{1:t-1}$ *seen classes* **do**
  Samples $\hat{s}^c$ according to classes c in $\mathcal{M}_s$.
  if c *is in* $\mathcal{C}^t$ *current classes* **then**
    Samples $\mathcal{T}_t^c$ according to classes c in $\mathcal{T}_t$.
    $p^c = \frac{\beta}{|\mathcal{F}|} \sum_{h \sim \mathcal{F}} f_\theta(h(\hat{s}^c)) + \frac{1-\beta}{|\mathcal{T}_t^c|} \sum_{(x;\cdot) \sim \mathcal{T}_t^c} f_\theta(x)$.
  else
    $p^c = \frac{1}{|\mathcal{F}|} \sum_{h \sim \mathcal{F}} f_\theta(h(\hat{s}^c))$.
// Prototypical Loss
$\mathcal{L}_{\texttt{cur\_pro}} = 0$.
$\mathcal{L}_{\texttt{pre\_pro}} = 0$.
for c *in* $\mathcal{C}^t$ *seen classes* **do**
  if c *is in* $\mathcal{C}^t$ *current classes* **then**
    for $(x; y)$ *in* $\mathcal{T}_t$ *according to* c **do**
      $\mathcal{L}_{\texttt{cur\_pro}} \mathrel{+}= \frac{\exp(-\texttt{MSE}(f_\theta(x); p^c))}{\sum_{l' \neq l} \exp(-\texttt{MSE}(f_\theta(x); p^{l'}))}$.
  else
    for $(s; y)$ *in* $\mathcal{M}_s$ **do**
      $\mathbf{z} = \{f_\theta(h(s)) | \forall h \in \mathcal{F}\}$
      $\mathcal{L}_{\texttt{pre\_pro}} \mathrel{+}= \frac{\exp(-\texttt{MSE}(\mathbf{z}; p^c))}{\sum_{l' \neq l} \exp(-\texttt{MSE}(\mathbf{z}; p^{l'}))}$.
return $\mathcal{L}_{\texttt{total}} = \mathcal{L}_{\texttt{cur\_pro}} + \alpha_1 \mathcal{L}_{\texttt{pre\_pro}} + \alpha_2 \mathcal{L}_{\texttt{task\_pre}} + \alpha_3 \mathcal{L}_{\texttt{task\_cur}}$

Algorithm 6: Prototypical Exemplar Generation

Input: Feature extractor f_θ, classifier f_ϕ, current task $\mathcal{T}_t$, $y_i^t \in \mathcal{Y}$, synthetic data $s = \{s^c\}_{l=1}^{\mathcal{C}^t}$ and optimizer $\texttt{opt}(s)$, coefficient α.

Output: θ^{t+T};

// Current Exemplar Alignment

for *c in current classes of* $\mathcal{T}_t$ **do**

  Samples $\mathcal{T}_t^c$ according to classes c in $\mathcal{T}_t$.

  $p^c = \frac{1}{n^c} \sum_{(x,y)\sim\mathcal{T}_t^c} f_\theta(x)$.

  $\mathcal{L}_1 \mathrel{+}= \texttt{MSE}(f_\theta(s^c); p^c)$.

// Previous Exemplar Alignment

for *c in* $\mathcal{C}^{1:t-1}$ *seen classes* **do**

  if *c is in current classes* **then**

    Samples $\hat{s}^c$ according to classes c in $\mathcal{M}_s$.

    $\mathcal{L}_2 \mathrel{+}= \texttt{MSE}(f_\theta(s^c); f_\theta(\hat{s}^c))$.

// Representation Shift Alignment

for *c in* $\mathcal{C}^{1:t-1}$ *seen classes* **do**

  if *c is in current classes* **then**

    Samples $\hat{s}^c$ according to classes c in $\mathcal{M}_s$.

    Samples $\hat{p}^c$ according to classes c in $\mathcal{M}_p$.

    $\mathcal{L}_{\texttt{shift}} \mathrel{+}= \texttt{MSE}(f_\theta(s^c); \hat{p}^c)$.

return $\mathcal{L}_{\texttt{sync\_proto}} = \alpha\mathcal{L}_1 + (1-\alpha)\mathcal{L}_2$

E. Experimental Settings

Training Details. Following existing works [7], we adopt ResNet-18 for S-CIFAR-100, and ResNet-50 for S-TinyImageNet, S-ImageNet-1K. For fair comparison, all methods use the same backbone without pretraining and optimizer. The optimizer is performed using Adam optimizer with $\beta_1 = 0.9$ and $\beta_2 = 0.999$. The training epochs vary across datasets: 50 epochs for CIFAR100, 100 epochs for S-TinyImageNet, and 100 for S-ImageNet-1K. We maintain a consistent batch size of 128 across all experiments. Results are averaged over 3 independent runs, and we report the corresponding standard deviation.

We present the detailed hyper-parameter setting of PEOCL in Table 7. These hyperparameters are carefully tuned to balance memory efficiency and performance, reflecting the varying complexity of the datasets. The hyperparameter settings of baseline methods are following existing works. For all datasets, we employ minimal data augmentation, consisting of random resized cropping to 224×224 pixels and random horizontal flipping during training, without any additional augmentation techniques. To prevent overfitting, we set the temperature parameter in the cross-entropy loss to 3 for all datasets. All experiments were conducted on NVIDIA RTX 4090 GPUs with 24GB memory using PyTorch 3.9.

Type	Hyperparameters	S-CIFAR-100 [15]	S-CIFAR-100 [15]	S-TinyImageNet [25]	S-ImageNet-1K [15]
Proto Network	Optimizer	Adam	Adam	Adam	Adam
	LR	0.05	0.05	0.05	0.05
	Temperature	0.1	0.1	0.1	0.1
	Alignment Coeff.	0.1	0.1	0.1	0.1
	Proto Coeff.	0.95	0.95	0.95	0.95
Synthetic Generator	Iterations	50	50	50	50
	Optimizer	AdamW	AdamW	AdamW	AdamW
	Initial Weight	10^{-4}	10^{-4}	10^{-4}	10^{-4}
	LR	0.1	0.1	0.1	0.1
	LR Scheduler	Cosine Annealing	Cosine Annealing	Cosine Annealing	Cosine Annealing

Table 7. Hyperparameter settings for different datasets.

Evaluation Metrics. We evaluate our method using two widely adopted metrics in the CL [7]. The first metric is the final accuracy, denoted as A_T, which measures the model's overall performance on all tasks after completing the entire training sequence. The second metric is the average accuracy, denoted as $\bar{A}$, which reflects the model's learning stability over time. It is computed as $\bar{A} = \frac{1}{T}\sum_{t=1}^{T} A_{\tau,t}$, where T is the total number of tasks and $A_{\tau,t}$ is the accuracy on task t after learning task τ. Together, these metrics quantify both the model's ability to acquire new knowledge and its capability to retain previously learned information, providing a comprehensive evaluation of continual learning performance.

F. Additional Experiments

F.1. Task Incremental Learning

Type	Method	Buffer SPCs	S-CIFAR-100 [15] T=10		S-CIFAR-100 [15] T=50		S-TinyImageNet [25] T=20		S-ImageNet-1K [15] T=100	
			$\overline{A}$	A_T	$\overline{A}$	A_T	$\overline{A}$	A_T	$\overline{A}$	A_T
Efficient Memory	CSReL	20	43.86±1.67	85.81±1.23	62.97±1.12	85.35±1.71	25.68±1.89	70.49±1.36	14.46±2.26	**67.68**±1.62
	BCSR	20	50.95±1.38	86.54±1.98	63.48±0.93	**89.92**±2.30	26.54±1.26	71.46±1.63	13.97±2.17	65.86±2.49
	OCS	20	48.85±0.88	85.64±1.61	61.24±1.22	86.37±1.49	23.84±0.63	68.46±1.94	13.57±0.42	66.92±0.61
	PBCS	20	**53.36**±1.06	**91.52**±2.03	**64.07**±1.13	**90.57**±1.22	**28.75**±0.67	**77.09**±2.31	16.14±1.21	**70.46**±2.45
	OnPro	20	47.51±1.26	84.41±1.83	60.75±1.78	85.03±2.17	22.78±1.18	70.49±2.06	11.28±0.88	65.27±0.33
Replay based	iCaRL	20	43.42±0.68	71.46±1.61	58.94±0.26	82.12±2.29	20.34±0.84	67.86±1.71	13.77±0.89	60.48±1.21
	DER	20	39.49±1.52	80.11±1.56	59.36±1.13	81.54±0.30	21.90±1.81	69.23±1.16	13.06±0.42	62.87±0.18
	DER++	20	41.75±0.16	81.26±0.28	60.77±1.71	85.79±1.78	22.06±1.36	70.29±1.54	12.37±1.36	64.55±4.21
	C-Flat	20	44.61±0.74	75.13±0.90	60.18±1.25	84.03±1.12	23.97±0.90	69.59±1.27	13.81±1.17	61.46±1.24
Replay free	LDC	20	37.12±0.88	68.92±0.61	55.61±1.26	75.30±1.21	16.61±0.64	63.17±0.74	12.48±1.04	60.87±1.21
	FeCAM	20	36.67±1.34	65.14±1.90	53.97±1.25	69.04±1.12	15.47±0.37	57.16±1.23	11.94±1.18	55.78±1.94
	ADC	20	38.35±0.80	69.88±2.04	56.20±1.57	73.58±1.67	17.42±0.58	64.85±1.62	12.91±1.17	62.48±0.98
Ours	ProtoCore	1	39.46±1.54	86.13±0.92	55.28±1.13	87.51±1.75	17.42±1.42	71.28±1.94	15.68±1.75	66.38±1.43
	ProtoCore	5	51.12±1.41	86.85±1.57	63.25±0.93	88.49±1.03	27.44±1.24	**72.45**±1.77	**18.07**±0.89	66.46±1.33
	ProtoCore	20 + 1	**58.46**±1.58	**87.15**±0.91	**67.91**±1.18	88.15±1.59	**34.46**±1.63	71.89±1.46	**22.18**±1.34	65.70±1.83

Table 8. Average ($\overline{A}$) and final (A_T) accuracy (%) comparison on benchmarks with total number of tasks T. Results are averaged over 10 runs with mean ± standard deviation. **Best** and **Second Best** results are highlighted.

F.2. Additional Computation and Memory Usage

Table 9. Comparison on S-ImageNet-1K (100 tasks) in terms of computation (Hours), GPU and memory usage (GB).

Method	**Comp. Time (Hours)**	**GPU Mem. (GB)**	**HDD Mem. (G)**
CSReL	214.61	10.04	12.08
BCSR	287.84	24.89	12.10
OCS	313.44	27.55	12.94
PBCS	391.63	18.89	12.14
OnPro	316.21	11.70	12.92
iCaRL	38.75	10.13	12.92
DER	65.79	12.56	12.73
DER++	83.40	23.61	12.59
C-Flat	105.03	19.48	12.12
LDC	73.20	50.50	0.47
FeCAM	50.28	13.45	0.00
ADC	45.43	57.14	0.47
ProtoCore	252.69	30.57	1.15

F.3. Time complexity.

The primary difference between our method and prior work is that we perform synthetic example generation at the end of every task. Unlike generative memory approaches that optimize full network weights, our method optimizes only the learnable quantized latent vector z, which has low dimensionality. Consequently, the optimization required for synthetic generation is substantially faster. To quantify this, we measure the wall-clock time consumed by the synthetic generation procedure per class on S-CIFAR-100, S-TinyImageNet, S-ImageNet-1K and report both an empirical per-class time and a simple complexity model.

Table 10. Analysis on time complexity of the synthetic example generation at the end of every task.

Method	**S-CIFAR-100 (T=10)** **Seconds/task**	**S-ImageNet-1K (T=100)** **Seconds/task**
PBCS	269.28	11112.84
CSReL	151.60	4740.84
BCSR	218.16	7368.84
OCS	235.31	8304.84
ProtoCore	161.85	6108.84