A THE USE OF LLMS

We used ChatGPT-4o to polish our manuscript, using the following prompt:

```
I want you to act as an expert in scientific writing.  I will
provide you with some paragraphs in English and your task is
to improve the spelling, grammar, clarity, conciseness, and
overall readability of the text provided, while breaking down long
sentences, reducing repetition and increasing logic.  You should use
artificial intelligence tools, such as natural language processing,
rhetorical knowledge, and your expertise in effective scientific
writing techniques to reply.  Provide the output as a table in
a readable mode.  The first column is the original sentence, the
second column is the sentence after editing, and the third column
provides explanation of your edits and reasons.  Please edit the
following text in a scientific tone:
```

B THEORETICAL ANALYSIS

B.1 ANALYSIS OF OPTIMIZATION FUNCTIONS

The optimization problem is given by:

$$\min_{W} \|KW - V\|_F^2 + \|\Gamma(W - W_0)\|_F^2 \tag{8}$$

where Γ is a diagonal matrix defined as:

$$\Gamma = \text{diag}(s_1^{1/4}, s_2^{1/4}, \ldots, s_d^{1/4}), \quad s_i = \sum_j k_{ij}^2 \tag{9}$$

To find the optimal solution, we take the first-order derivative and set it equal to zero:

$$K^T(KW - V) + \Gamma^2(W - W_0) = 0 \tag{10}$$

$$\Rightarrow (K^T K + \Gamma^2)(W - W_0) = K^T V - K^T K W_0 \tag{11}$$

$$\Rightarrow \Delta W = (K^T K + \Gamma^2)^{-1} K^T (V - K W_0) \tag{12}$$

B.2 MATRIX UPDATING RULE

We have the following expressions for K and V:

$$K = \begin{bmatrix} K_1 \\ K_2 \\ \vdots \\ K_T \end{bmatrix}, \quad V = \begin{bmatrix} V_1 \\ V_2 \\ \vdots \\ V_T \end{bmatrix}$$

For $K^T K$, we can express it as:

$$K^T K = \sum_{t=1}^{T} K_t^T K_t \tag{13}$$

For $K^T V$, we have:

$$K^T V = \sum_{t=1}^{T} K_t^T V_t \tag{14}$$

B.3 Theoretical Analysis of the Auto-Balancing Mechanism

The objective function is formally defined as:

$$\min_W \|(KW - V)\|_F^2 + \|\Gamma(W - W_0)\|_F^2, \tag{15}$$

where the diagonal matrix Γ is set as $\Gamma_{ii} = s_i^{1/4}$, with $s_i = \sum_t k_{ti}^2$ representing the accumulated feature energy of the input key matrix K. In practice, the raw feature energy s_i inherently exhibits extreme variations across different feature dimensions and tasks (e.g., varying from 10^4 to 10^6 in our CDFSOD 5-shot experiments). This creates a severe scale imbalance for optimization: a fixed scalar λ would be either too negligible for high-energy dimensions or too restrictive for low-energy ones. To resolve this, we derived our Γ design based on the optimization curvature to automatically normalize these scales.

Feature-wise Adaptivity via Curvature Analysis. We address the optimization imbalance by analyzing the Hessian (curvature) of the objective function $\mathcal{L}$.

- **Data Term Curvature:** The Hessian of the first term $\|KW - V\|_F^2$ with respect to W is $2K^T K$. Suppose the input feature K has a magnitude scale M_k. The curvature of this term scales quadratically, i.e., $\mathcal{O}(M_k^2)$. This creates extremely steep optimization landscapes for high-energy features, leading to dominance in updates if not balanced.
- **Adaptive Regularization:** The Hessian of our regularization term is $2\Gamma^2$. Since the feature energy s_i scales with $\mathcal{O}(M_k^2)$, our design $\Gamma_{ii} = s_i^{1/4}$ implies that Γ^2 scales linearly with $\mathcal{O}(M_k)$ (as $\Gamma^2 \propto \sqrt{s_i} \propto M_k$).

This formulation achieves a critical balance mechanism:

- **Adaptivity:** The regularization strength Γ^2 grows with feature magnitude ($\mathcal{O}(M_k)$), providing necessary constraints for strong features compared to fixed scalars.
- **Plasticity:** The regularization grows slower than the data term ($\mathcal{O}(M_k)$ vs. $\mathcal{O}(M_k^2)$). This "sub-quadratic" scaling prevents the regularization from becoming over-rigid (which would happen if $\Gamma_{ii} = s_i^{1/2}$, where $\Gamma^2 \sim \mathcal{O}(M_k^2)$), thereby allowing sufficient injection of new knowledge even for high-signal features.

C Experiment

In our experiments, we primarily adopt **Grounding DINO** (Liu et al., 2024), a state-of-the-art open-vocabulary object detector that unifies grounding with strong detection performance. We also evaluate on **GLIP** (Li et al., 2022b), a vision-language pre-training model that aligns detection with phrase grounding, in order to verify the generality of our auto-balance strategy across different models and task scales.

For fine-tuning, we update only the output layers of all FFN modules. The batch size is set to 8 for Grounding DINO and 2 for GLIP, and we use the AdamW optimizer with a learning rate of 1×10^{-4}. Training is conducted for 18 epochs on most datasets, with slight adjustments under different shot settings to achieve better adaptation.

C.1 Training Details

CDFSOD. The Cross-Domain Few-Shot Object Detection (CDFSOD) benchmark (Fu et al., 2024) uses MS-COCO (Lin et al., 2014) as the source domain and includes six heterogeneous target domains: ArTaxOr, Clipart1K, DIOR, DeepFish, NEU-DET, and UODD. These domains differ substantially in visual style, resolution, and object distribution, providing a challenging setup for cross-domain transfer.

Table 11: Few-shot results on **CDFSOD** 5-shot using **Grounding DINO-B**. The base model is the pre-trained Grounding DINO-B without fine-tuning. Methods marked with † denote the final model obtained by sequential fine-tuning on a single model, evaluated across all datasets.

Method	ArTaxOr	Clipart1K	DeepFish	DIOR	NEU-DET	UODD	Average	COCO
Base model	12.8	49.1	28.6	4.5	1.2	10.1	17.7	59.7
FFN W_{out} only	69.2	60.2	35.1	29.4	22.2	21.5	39.6	–
Fully fine-tune	70.3	59.3	37.0	29.4	22.3	24.3	40.5	–
Fully fine-tune†	52.4	51.1	35.2	22.6	19.8	22.1	33.9	47.9

ODinW-13. ODinW-13 (Li et al., 2022a) is a subset of the ELEVATER benchmark (Li et al., 2022a), consisting of 13 diverse object detection tasks drawn from various open-world datasets. The benchmark emphasizes robustness to domain shifts and serves as a standard evaluation for open-vocabulary detectors.

Few-shot setting. Together, CDFSOD and ODinW-13 form 19 few-shot tasks with clear distribution shifts, on which open-vocabulary detectors typically experience performance degradation resembling out-of-distribution scenarios. We follow the standard K-shot protocol with $K \in \{1, 5, 10, 30, 50\}$, where K labeled images per class are sampled as the support set and the remaining images are used for evaluation. For fair comparison, all baseline methods are consistently applied to the output layers across all FFN modules of the model.

EWC. For Elastic Weight Consolidation (EWC) (Kirkpatrick et al., 2017), we introduce a quadratic penalty on parameter updates, weighted by the Fisher information estimated from the previous task. The regularization coefficient is set to $\lambda_{\text{EWC}} = 10000$. We use the AdamW optimizer with a learning rate of 1×10^{-4} and a batch size of 8, training for 18 epochs on most tasks. The overall objective can be written as

$$\mathcal{L}(\theta) = \mathcal{L}_{T+1}(\theta) + \frac{\lambda_{\text{EWC}}}{2} \sum_t \sum_i F_{t,i} \left(\theta_i - \theta^*_{t,i}\right)^2, \tag{16}$$

where $\mathcal{L}_{T+1}(\theta)$ denotes the standard loss on the current task, $F_{t,i}$ represents the Fisher information of parameter θ_i estimated from task t, and $\theta^*_{t,i}$ is the optimal parameter value obtained after training on task t. This regularization term penalizes large deviations from previously important parameters, thereby mitigating catastrophic forgetting across sequential tasks.

Adam-NSCL. For Adam-NSCL (Null-Space Continual Learning with Adam) (Wang et al., 2021), we constrain parameter updates to the approximate null space of previously learned features, following the SVD-based formulation. Specifically, we select the null-space basis U_2^l associated with the smallest singular values of Λ_2^l, and adaptively choose Λ_2^l with diagonal entries $\lambda \in \{\lambda \mid \lambda \leq a\lambda^l_{\min}\}$, where $\lambda^l_{\min}$ is the smallest singular value. In our experiments, we set the hyperparameter $a = 40$. Optimization is performed using the Adam optimizer with a learning rate of 1×10^{-4}, a batch size of 8, and training for 18 epochs on most tasks.

SD-LoRA. For SD-LoRA (Spectral Decay LoRA) (Wu et al., 2025), we adopt the same adapter structure as LoRA but additionally apply spectral decay regularization on the low-rank updates to improve stability across sequential tasks. The hyperparameters are set as follows: rank $r = 16$, initial spectral decay coefficient $\alpha_{\text{init}} = 1.0$, dropout rate 0.0, LoRA learning rate 0.002, and LoRA weight decay 0.0. Training is performed for 18 epochs on most tasks with a batch size of 8.

Training Details for CLIP Experiments. We evaluated our method on 11 diverse classification datasets, organized in the following order: Aircraft, Caltech101, CIFAR100, DTD, EuroSAT, Flowers102, Food101, MNIST, OxfordPets, StanfordCars, and SUN397. For optimization, we standardized the training duration to 500 iterations per dataset. Regarding the scope of parameter updates, our ABME method specifically targets the output projection layers of all FFN blocks within the ViT-B/16 (Dosovitskiy, 2020) encoder, whereas the baseline methods (ZSCL (Zheng et al., 2023b) and WiSE-FT (Wortsman et al., 2022)) involve full-model fine-tuning.

Table 12: Performance comparison of different methods under various corruptions on CDFSOD 5-shot. All values represent mean Average Precision (AP) in percentage. Note that **Oracle** is obtained by independent fine-tuning on each corrupted dataset (averaged over 90 total datasets). Continual learning setting follows a sequential training protocol according to the table rows; specifically, within each corruption, it trains on ArTaxor, Clipart1k, DeepFish, DIOR, NEU-DET, and UODD sequentially.

Corruption	Base	Adam-NSCL	Ours	Oracle
gaussian_noise	8.6	18.0	21.6	23.0
shot_noise	8.1	18.1	21.6	23.1
impulse_noise	8.7	18.4	22.1	23.4
defocus_blur	11.6	23.0	25.7	28.2
glass_blur	11.4	24.2	27.1	30.3
motion_blur	9.3	20.5	22.7	24.3
zoom_blur	2.7	6.2	6.9	10.7
snow	9.6	20.0	23.0	24.9
frost	10.1	20.3	22.9	24.2
fog	14.2	26.7	29.6	32.4
brightness	12.9	25.5	28.8	31.6
contrast	8.7	20.5	21.8	26.2
elastic_transform	12.0	24.9	28.6	33.3
pixelate	12.2	24.5	28.2	31.6
jpeg_compression	11.5	22.5	25.9	28.2
Average	**10.1**	**20.8**	**23.8**	**26.4**
COCO	**59.7**	**57.2**	**57.5**	**–**

D Additional Baseline Comparisons and Resource Efficiency

D.1 Comparison with State-of-the-Art Methods

To further validate the effectiveness of ABME, we included two additional state-of-the-art continual learning baselines: SGP (Saha & Roy, 2023) and SVFCL (Wang et al., 2025). We evaluated all methods under the CDFSOD 5-shot setup. As shown in Table 13, ABME significantly outperforms these recent approaches.

Table 13: Extended comparison on CDFSOD 5-shot. We incorporated two additional baselines: SGP and SVFCL. ABME achieves the best trade-off between plasticity (New-task mAP) and stability (Old-task mAP).

Method	New-task mAP ↑	Old-task mAP ↑	RR ↑	AGR ↑
EWC (2017)	31.0	57.1	95.6%	78.3%
Adam-NSCL (2021)	29.1	57.8	96.8%	73.5%
SGP (2023)	32.6	46.1	77.2%	82.3%
SD-LoRA (2025)	24.9	54.2	90.8%	62.9%
SVFCL (2025)	33.2	52.9	88.6%	83.8%
ABME (Ours)	**38.5**	**57.0**	**95.5%**	**97.2%**
Oracle	39.6	59.7	–	–

D.2 Resource Consumption Analysis

We compared the training time, GPU memory usage, and extra storage cost of all methods on the CDFSOD 5-shot setup. All models were trained on the same hardware configuration. As detailed in Table 14, ABME maintains **constant resource efficiency** comparable to standard FFN fine-tuning, requiring only a fixed $\sim$1.3 GB storage independent of the task count. In contrast, baselines exhibit varying overheads: EWC accumulates storage costs as tasks are added ($0.7 \rightarrow 3.4$ GB), while Adam-NSCL requires increased memory for subsequent tasks compared to the initial stage

(1640 → 1731 MB). SD-LoRA suffers from linear memory growth and the longest training time. Furthermore, our final optimization step (solving Eq. 5) typically completes in <**10 seconds**, incurring negligible operational overhead.

Table 14: Resource consumption analysis on CDFSOD 5-shot. For metrics formatted as "Start → End" (e.g., 932 → 977), the values correspond to the consumption recorded during the training of the **first task** and the **final task**, respectively, indicating the variation in resource usage as tasks accumulate.

Method	Training Time (s)	GPU Memory (MB)	Extra Storage (GB)	CDFSOD (New)	COCO (Old)
SD-LoRA	1121	932 → 977	–	24.9	54.2
EWC	1044	1640	0.7 → 3.4	31.0	57.1
Adam-NSCL	904	1640 → 1731	1.1	29.1	57.8
ABME (Ours)	1029	1640	1.3	38.5	57.0
Oracle (FFN-out)	1024	1640	–	39.6	–

E Core Implementation Code

We provide the core PyTorch implementation of the Auto-Balanced Model Editing (ABME) algorithm below. For clarity, we present the solver for a single layer. The input `stats` dictionary contains the accumulated sufficiency statistics extracted from the support set:

- `kk`: The autocorrelation matrix of keys $K^\top K \in \mathbb{R}^{d_{in} \times d_{in}}$.
- `kv`: The cross-correlation matrix $K^\top V \in \mathbb{R}^{d_{in} \times d_{out}}$.
- `sum_x`, `sum_y`: Sum of input keys and output values (for bias handling).
- `n`: Total number of support samples.

```python
import torch

@torch.no_grad()
def solve_abme_layer(W_old, b_old, stats, eps=1e-8):
    """
    Apply ABME update to a single linear layer (FFN output).

    Args:
        W_old: Original weights [out_dim, in_dim]
        b_old: Original bias [out_dim]
        stats: Dictionary containing K^T K, K^T V, etc.
    """
    out_dim, in_dim = W_old.shape

    # ------------------------------------------------------------
    # 1. Load Statistics
    # ------------------------------------------------------------
    # kk: K^T K [in, in], kv: K^T V [in, out]
    kk = stats['kk'].to(W_old.dtype)
    kv = stats['kv'].to(W_old.dtype)
    sx = stats['sum_x'].to(W_old.dtype).reshape(in_dim, 1)
    sy = stats['sum_y'].to(W_old.dtype).reshape(1, out_dim)
    n  = torch.tensor(stats['n'], dtype=W_old.dtype)

    # ------------------------------------------------------------
    # 2. Construct Augmented System (incorporating bias)
    # ------------------------------------------------------------
    # A corresponds to the augmented K^T K matrix
    A = torch.zeros((in_dim + 1, in_dim + 1), dtype=W_old.dtype)
    A[:in_dim, :in_dim] = kk
    A[:in_dim, in_dim:] = sx
    A[in_dim:, :in_dim] = sx.t()
    A[in_dim, in_dim]   = n
```

```
    # B corresponds to the augmented K^T V matrix
    B = torch.zeros((in_dim + 1, out_dim), dtype=W_old.dtype)
    B[:in_dim, :] = kv
    B[in_dim:, :] = sy

    # ------------------------------------------------------
    # 3. Prepare Original Parameters [W0; b0]
    # ------------------------------------------------------
    W0_aug = torch.cat([W_old.t(), b_old.reshape(1, out_dim)], dim=0)

    # ------------------------------------------------------
    # 4. Compute Auto-Balanced Regularization Matrix (Gamma^2)
    # ------------------------------------------------------
    # Get diagonal energy s_i = (K^T K)_ii
    s = kk.diag()

    # Numerical stability
    floor = s.mean() * eps
    s = torch.clamp(s, min=floor)

    # Our design: Gamma = diag(s^{1/4}) => Gamma^2 = diag(sqrt(s))
    P = torch.zeros((in_dim + 1,), dtype=W_old.dtype)
    P[:in_dim] = s.sqrt()      # Regularization for weights
    P[in_dim]  = torch.sqrt(n) # Regularization for bias

    # ------------------------------------------------------
    # 5. Solve the Linear System
    # ------------------------------------------------------
    # Objective: (A + Gamma^2) W* = (B + Gamma^2 W0)
    A_reg = A + torch.diag(P)           # LHS Matrix
    RHS   = B + W0_aug * P.unsqueeze(1) # RHS Term

    # Solve using Cholesky or LU (torch.linalg.solve)
    W_new_aug = torch.linalg.solve(A_reg, RHS)

    # ------------------------------------------------------
    # 6. Extract Updated Weights and Bias
    # ------------------------------------------------------
    W_new = W_new_aug[:in_dim, :].t() # [out_dim, in_dim]
    b_new = W_new_aug[in_dim, :]      # [out_dim]

    return W_new, b_new
```

Listing 1: PyTorch implementation of the ABME update for a single layer.