A. Additional Technical Details

A.1. Summary of Notaitons

Tab. 1 summarizes the major notations.

A.2. Estimation of $M_w(\boldsymbol{s})$

This section provides the details for estimating $M_w(\boldsymbol{s})$ in § 4.3.1. $M_w(\boldsymbol{s})$ captures the memory overhead introduced by all LoRA modules selected under strategy $\boldsymbol{s}$, including their parameters, gradients, and optimizer states. Let $M_{param}(\boldsymbol{s})$ be the total parameter memory of the inserted modules. Gradients require the same amount of memory as parameters. The optimizer state size depends on the optimizer. For example, AdamW maintains two moment estimates per parameter, resulting in an additional $2 \times M_{param}(\boldsymbol{s})$ memory cost [31]. Therefore, for AdamW, the total module-related memory is $M_w(\boldsymbol{s}) = 4 \cdot M_{param}(\boldsymbol{s})$, where the factor 4 accounts for parameters, gradients, and two sets of moment estimates. In general, $M_w(\boldsymbol{s})$ scales linearly with $M_{param}(\boldsymbol{s})$ and can be computed directly once the optimizer is specified.

A.3. Complete Intra-Block Dependencies for $g_x(k)$

This section explains how to configure $g_x(k)$ for the remaining five intra-block dependencies in § 4.3.2.

Inserting a module w_k at W_k (Fig. 5a). The forward pass is $K = x_{\text{in}}(W_k + w_k)$. To compute $\nabla_{w_k} L$, gradients flow from $\nabla_{\text{Attn}(Q,K,V)} L$ to $\nabla_K L$ via the attention operator. Since $\text{Attn}(Q,K,V) = \text{Softmax}(\frac{QK^\top}{\sqrt{d}})V$, the dependency between K and the output involves the interaction with Q and V. Therefore, computing $\nabla_K L$ requires the stored activations Q and V. Finally, $\nabla_{w_k} L$ needs the input activation x_{in}. Thus, insertion location at W_k requires storing Q, V, and x_{in}.

Inserting a module w_v at W_v (Fig. 5b). The forward pass is $V = x_{\text{in}}(W_v + w_v)$. To compute $\nabla_{w_v} L$, gradients flow to $\nabla_V L$ through the attention operator. In the term $\text{Attn}(Q,K,V) = \text{Softmax}(\frac{QK^\top}{\sqrt{d}})V$, the variable V is multiplied by the attention weights (the result of Softmax). Consequently, the backward pass to V requires the reconstruction of these attention weights, which depends on the stored activations Q and K. Calculating $\nabla_{w_v} L$ further requires the input x_{in}. Thus, insertion location at W_v requires storing Q, K, and x_{in}.

Inserting a module w_o at W_o (Fig. 5c). The forward pass is $x^1_{\text{out}} = (W_o + w_o) \cdot \text{Attn}(Q,K,V) + x_{\text{in}}$. To compute $\nabla_{w_o} L$, gradients flow directly from $\nabla_{x^1_{\text{out}}} L$. The local gradient computation for the weight w_o requires the input to the linear projection, which is the output of the attention mechanism, $\text{Attn}(Q,K,V)$. Thus, insertion location at W_o requires storing the activation vector $\text{Attn}(Q,K,V)$.

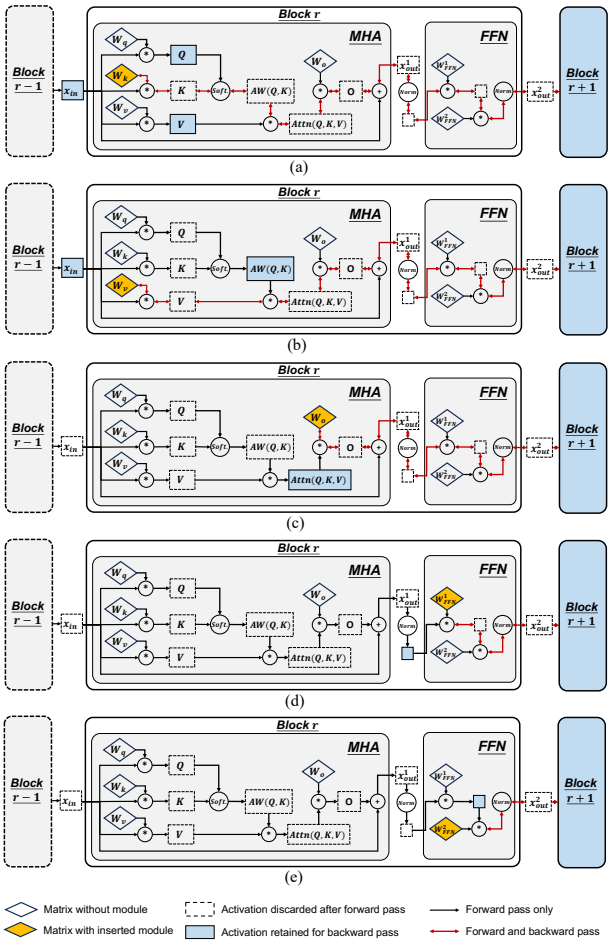


Figure 1. Intra-block activation dependency analysis. (a)-(e) demonstrate the activation preservations when inserting modules into W_k, W_v, W_o, W^1_{FFN}, and W^2_{FFN} matrices, respectively.

Inserting a module w^1_{FFN} at W^1_{FFN} (Fig. 5d). Let the FFN be defined as $FFN(x) = W^2_{\text{FFN}} \cdot \psi((W^1_{\text{FFN}} + w^1_{\text{FFN}}) \cdot \text{Norm}(x))$, where ψ is the activation function like ReLU. To compute $\nabla_{w^1_{\text{FFN}}} L$, the backpropagation process requires the input to this specific layer. The input to W^1_{FFN} is the normalized output of the attention block, denoted as $\text{Norm}(x^1_{\text{out}})$. Thus, insertion at W^1_{FFN} requires storing $\text{Norm}(x^1_{\text{out}})$.

Inserting a module w^1_{FFN} at W^2_{FFN} (Fig. 5e). The forward pass involves $x^2_{\text{out}} = (W^2_{\text{FFN}} + w^2_{\text{FFN}}) \cdot v + x^1_{\text{out}}$, where $v = \psi(W^1_{\text{FFN}} \cdot \text{Norm}(x^1_{\text{out}}))$ is the intermediate activation of the feed-forward network. To compute $\nabla_{w^2_{\text{FFN}}} L$, the gradient computation requires the input to the second linear layer, which is the intermediate hidden state v. Thus, insertion at W^2_{FFN} requires storing the intermediate FFN activation v.

A.4. Pseudocode for DP-based Scheduler

Algorithm 1 illustrates the workflow of the module insertion scheduler. The DP state space is $O(|\mathcal{K}| \cdot U)$. Com-

Notation	Definition
$t_1, \ldots, t_n$	Previously learned tasks
t_{n+1}	New task to be fine-tuned
d	Batched input data of the new task
θ	Frozen pre-trained backbone parameters
$\mathcal{K}$	Set of projection matrices, i.e. candidate LoRA module insertion locations
W_k	Frozen projection matrix at backbone location $k \in \mathcal{K}$
w_i^k	LoRA module parameters for task t_i at location k
$\boldsymbol{s}$	Module insertion strategy (vector where s^k is the rank at location k)
M_B	Total memory budget for LoRA fine-tuning
$\boldsymbol{\lambda}_{n+1}$	Predicted module importance vector for new task t_{n+1}
λ_{n+1}^k	Importance of module at location k for new task t_{n+1}
λ_i^k	Importance of module at location k for learned task t_i
$\phi(t_{n+1}, t_i)$	Task similarity between new task t_{n+1} and learned task t_i
$x(w_i^k; d)$	Output activations of LoRA module w_i^k on data d
$x(W_k; d)$	Output activations of frozen matrix W_k on data d
$x(\theta + \boldsymbol{w}_i; d)$	Output activations of the backbone θ merged with task modules $\boldsymbol{w}_i$
$x^*(\theta + \boldsymbol{w}_{n+1}; d)$	Approximated activation of the new task
$M(\boldsymbol{s})$	Peak fine-tuning memory usage under strategy $\boldsymbol{s}$
M_{con}	Constant memory usage (frozen backbone parameters and runtime overhead)
$M_w(\boldsymbol{s})$	Memory usage of LoRA modules (parameters, gradients, optimizer states)
$M_x(\boldsymbol{s})$	Memory usage of cached activations during backpropagation
$\boldsymbol{g}_x(\boldsymbol{s})$	Binary mask indicating activations to be retained for strategy $\boldsymbol{s}$
$\boldsymbol{m}_x$	Memory size vector of candidate activations
$W_q, W_k, W_v, W_o, W_{FFN}^1, W_{FFN}^2$	Frozen projection matrices in backbone
r	Rank size of an inserted module
Q, K, V	Query, Key, and Value matrices in attention blocks
$P[k][M_b]$	Maximum cumulative importance at location k with memory cost $\leq M_b$
k_s	Index of the nearest previously inserted module location (predecessor)
ΔM	Incremental memory cost of inserting a module given predecessor k_s
$\mathcal{R}$	Set of candidate LoRA ranks
$h(r)$	Rank-based importance scaling function
U	Number of memory discretization bins

Table 1. Summary of major notations.

puting each state $P[k][M_b]$ involves iterating over all ranks r ($O(|\mathcal{R}|)$) and preceding locations k_s ($O(|\mathcal{K}|)$). Thus, the overall time complexity is $O(|\mathcal{K}|^2 \cdot |\mathcal{R}| \cdot U)$.

B. Additional Experimental Setups

B.1. Details of Baselines

The configurations of the baselines are as follows.

B.1.1. Zero-FT

- Prompt Tuning [14] adds prompt vectors to the keys and values of the attention module in each transformer block. We adopt a deep prompt tuning approach, incorporating 16 learnable prompt vectors per block into the first 24 blocks of the backbone following [20].
- AdapterSoup [3] averages the weight space of pre-trained domain adapters, selected via textual clustering or similarity, to enhance generalization to new domains.
- AdapterFusion [22] introduces a fusion layer with an attention mechanism to combine multiple pre-trained task adapters. We implement the fusion layer with a reduction factor of 16, balancing parameters and performance.
- LoraHub [11] applies a gradient-free evolutionary strategy to learn scalar weights for combining LoRA modules. We set the LoRA rank to $r = 8$ and randomly select 20 candidate modules for composition.

B.1.2. Non-LoRA-based PEFT

- Full tuning [12] updates all parameters of the pre-trained backbone and classification head.
- Linear tuning [12] freezes the pre-trained backbone and only updates the parameters of the final linear classification head.
- BitFit [28] optimizes the bias terms within all linear layers of each transformer block. We apply this optimization to every bias term throughout the network, following the approach outlined in the original implementation.
- RS-Bypass [13] creates an independent bypass network

Algorithm 1 DP-based Module Insertion Scheduler

Require: Memory budget M_{B}, Module importance scores λ_{n+1}^{k}, Memory prediction function $M(\boldsymbol{s})$, Rank set $\mathcal{R} = \{0, 2^0, \ldots, 2^6\}$.

Initialize DP table $P[k][M_b] \leftarrow -\infty$ for all k and M_b
Set $P[0][M_b] \leftarrow 0$ for all M_b
Set $U \leftarrow 500$
Set $\mathcal{R} \leftarrow [0, 1, 2, 4, 8, 16, 32, 64]$
Set the step size for memory bins $\Delta M_b = \frac{M_{\mathrm{B}}}{U}$
for $k = 1$ to $|\mathcal{K}|$ **do**
  for $M_b = 0$ to M_{B} with step ΔM_b **do**
    Initialize $P[k][M_b] \leftarrow P[k-1][M_b]$ {Start with the value from the previous state (no insertion)}
    for $r \in \mathcal{R}, r > 0$ **do**
      for $k_s = 0$ to $k - 1$ **do**
        $\Delta M \leftarrow M(\boldsymbol{s}_k) - M(\boldsymbol{s}_{k_s})$
        if $P[k_s][M_b - \Delta M] > -\infty$ **then**
          $imp \leftarrow P[k_s][M_b - \Delta M] + h(r)\lambda_{n+1}^{k}$ {Calculate the new importance if inserting rank r}
          if $imp > P[k][M_b]$ **then**
            $P[k][M_b] \leftarrow imp$
          end if
        end if
      end for
    end for
  end for
end for
$\boldsymbol{s} \leftarrow \max_{0 \le k \le |\mathcal{K}|} P[k][M_{\mathrm{B}}]$
Backtrack to find optimal module insertion strategy
return $\boldsymbol{s}$

using detached Res-Tuners to avoid backpropagation through the pre-trained backbone, reducing memory usage. We use a learnable fusion parameter initialized at 0.5, with other settings consistent with RS.

- Adapter [9] inserts adapters between transformer layers, consisting of a down-projection, followed by a nonlinear activation, and an up-projection. We set the bottleneck dimension $r = 128$.
- AdaptFormer [2] introduces a lightweight bottleneck module (AdaptMLP) alongside the frozen MLP block, consisting of down-projection and up-projection layers. We set the bottleneck middle dimension to $\hat{d} = 64$ and the scaling factor to $s = 0.1$.
- FacT-TK [15] tensorizes transformer weights into a 3D tensor and decomposes the weight increments into lightweight factors for efficient tuning. We typically use a rank of $r = 8$ for the decomposition factors.
- RS [13] separates parameter-efficient tuners from the backbone, treating them as residual modules added to frozen operations, allowing flexible combinations of adapters, prefixes, and prompts. We set the prompt/prefix length and adapter middle dimension to 10.
- ConvPass [14] inserts trainable convolutional bypass modules (1×1, 3×3, 1×1) in parallel to the attention and MLP blocks to introduce visual inductive bias. We set the bottleneck dimension to $h = 8$ for the convolutional layers.
- LISA [21] uses layerwise importance sampling to freeze most layers during optimization, updating only the embedding, head, and a few random intermediate layers. We configure the strategy to update the embedding layer, the head, and 2 random intermediate layers (E+H+2L).
- LoSA [20] adapts transformers by adding low-rank mixer blocks in parallel to the frozen backbone to efficiently process features. We set the low-rank adaptation rank to $r = 8$.
- HST [18] attaches a lightweight Hierarchical Side Network (HSN) to the frozen backbone to extract multi-scale features and uses a Meta-Register token to aggregate global information. We use a single trainable token for the Meta-Register and align the HSN stages with the backbone.
- UniPT [5] supports transfer learning via a parallel network with an interaction module for processing intermediate activations and a confidence aggregation module for cross-layer feature integration. We adopt a reduction factor of 4 for the parallel network dimensions.

B.1.3. LoRA-based PEFT

- LoRA [10] decomposes the weight matrices into trainable low-rank parameters to efficiently approximate weight updates. In our approach, we apply this adaptation to all linear projections in the transformer block (including query, key, value, output, and MLP), using a rank of $r = 8$.
- NOAH [32] treats Adapter, LoRA, and VPT as modular prompts and uses a one-shot neural architecture search to find the optimal module design and hyperparameters for each transformer block. We train the supernet for 500 epochs and perform the evolutionary search for 5 epochs.
- π-Adapter [26] uses Fisher Information Matrix embeddings to identify similar tasks and interpolates parameters from pre-trained lightweight experts for task adaptation. We use $k = 2$ auxiliary experts for interpolation and optimize the interpolation weights along with the target expert parameters.
- AdaLoRA [30] applies singular value decomposition to parameterize incremental updates and dynamically allocates the parameter budget by pruning singular values based on importance scores. We use a cubic schedule to prune the rank from an initial value to the target budget, with orthogonal regularization $\gamma = 0.1$.
- AutoLoRA [31] uses a meta-learning framework to determine layer-specific optimal ranks by learning selection

variables for decomposed rank-1 matrices. The selection variables start at a dimension of 8, with an alternating optimization strategy for updating model weights and selection variables.

B.2. Other Experimental Setups

Implementation Details. All experiments are conducted on 8 × NVIDIA GeForce RTX 3090 GPUs. Unless otherwise specified, we use the default configurations of the pre-trained foundation models, including the AdamW optimizer and an input resolution of 224×224 following [20, 26, 31]. When w_k^i is unavailable for a task in Eq. (3), we set $\lambda_k^i = 0$. Intuitively, if task t_i already performs well without a module at location k, then a similar new task is unlikely to require one there either.

Settings for Vision-Language Tasks. We strictly follow standard evaluation protocols for all VL tasks [26]. We report the standard accuracy metric (ACC@0.5) on validation and test splits. The training configuration includes 100 epochs, a dropout rate of 0.1, a warmup ratio of 0.06, a learning rate of $1e{-}4$, a batch size of 4, and a label smoothing factor of 0.1. For LoRA-based adaptation with fixed ranks, we set a default rank of $r = 8$.

Settings for Vision Tasks. For image classification, we train for 200 epochs with a batch size of 4 and a learning rate of $1e{-}4$. A label smoothing ratio of 0.1 is applied. Regarding data augmentation, we follow the protocols established in prior work [26], incorporating random resize cropping, random horizontal flipping, RandAug, and random erasing. Furthermore, we utilize Mixup and CutMix with alpha values set to 0.8 and 1.0, respectively, applied with an overall probability of 0.5 per batch.

Settings for Language Tasks. Following [26], we adopt their task-specific instructions. For hyperparameter optimization, we perform a grid search on the development set to identify the best configuration for each task. The search space includes training epochs $\{5, 7, 10\}$, learning rates $\{3e{-}2, 1e{-}4, 5e{-}5\}$, and batch sizes $\{8, 16, 32\}$.

C. Additional Experiments

C.1. Impact of Task Similarity

Settings. To evaluate the effectiveness of our task similarity prediction, we analyze how the predicted similarity between a learned task and a new task correlates with the accuracy gain obtained by transferring importance from that single learned task. For each new task, we consider each learned task in turn as the sole source and plot a point with its predicted similarity on the x-axis and the resulting accuracy improvement on the y-axis, as shown in Fig. 2.

The learned task set includes the five base tasks from § 5.2.1 and five additional tasks: video captioning (MSVD [1]), video question answering (MSVD-QA [27]), audio captioning (AudioCaps [16]), text-to-audio (WavCaps [19]), and text summarization (CNN/Daily Mail [7]). For new tasks, we use four representative benchmarks that cover both image and NLP modalities: CIFAR-100, VQA (test-std), MNLI, and ImageNet.

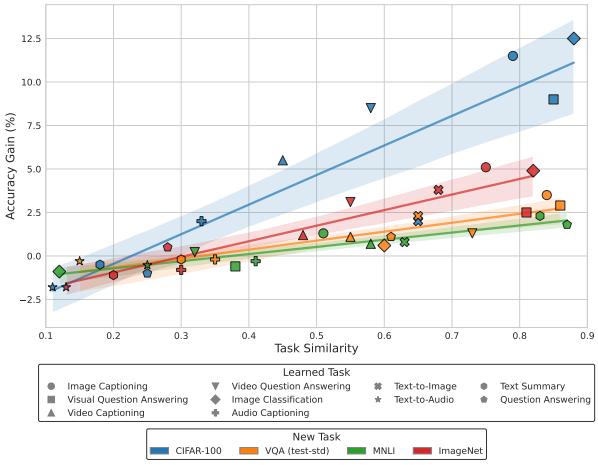


Figure 2. Impact of task similarity on fine-tuning accuracy. Each point denotes the accuracy gain when transferring module importance from a learned task (shape) to a new task (color).

Results. For each new task, we fit a linear model between predicted similarity and accuracy improvement and report the coefficient of determination R^2. The R^2 values for CIFAR-100, VQA, MNLI, and ImageNet are 0.82, 0.81, 0.84, and 0.86, respectively, indicating a strong positive correlation between task similarity and the benefit of importance transfer. In other words, more similar learned tasks tend to provide larger accuracy gains when used as sources for module-importance transfer.

We also make a few observations from Fig. 2. *(i)* Similar learned tasks yield larger gains for relatively simple new tasks than for more challenging ones. CIFAR-100 (blue line) exhibits the steepest slope: highly similar source tasks can significantly boost accuracy, while dissimilar sources can even degrade performance. In contrast, ImageNet (red line) has a gentler slope, likely because it depends on diverse visual features, making it less sensitive to any single source task. *(ii)* Second, transferring from the Text-to-Image task leads to a larger accuracy boost for ImageNet (3.8%) than for CIFAR-100 (2.1%), even though both are image classification tasks. This suggests that the richer visual-text alignment learned in Text-to-Image pre-training aligns better with the complex visual concepts required by ImageNet. *(iii)* When the similarity score between a learned task and a new task falls below 0.35, transferring importance from that learned task typically reduces the new task's accuracy. Motivated by this, in all experiments we re-

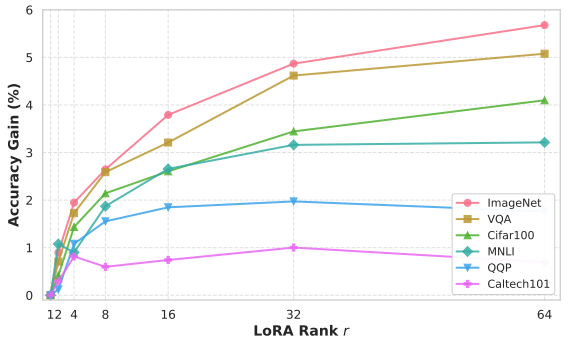


Figure 3. Accuracy gain (%) relative to $r = 1$ for varying LoRA ranks across different tasks.

Method	Function $h(r)$	Cifar100	ImgNet	Caltech101	MNLI	QQP	VQA	Avg
LoRA	$r = 8$	67.9	87.6	88.0	86.3	90.1	73.4	82.2
TaskIT	$h(r) \propto 1$	77.1	83.5	86.5	80.8	85.2	68.8	80.3
	$h(r) \propto r$	73.9	88.9	91.4	86.2	83.1	74.2	83.0
	$h(r) \propto \ln r$	80.8	88.7	91.0	85.0	**91.1**	74.3	85.2
	$h(r) \propto \sqrt{r}$	81.2	89.1	91.9	86.1	**91.5**	74.9	85.8
	$\boldsymbol{h(r) \propto \sqrt{\ln r}}$	**83.4**	**89.3**	**92.1**	**86.3**	89.6	**75.1**	**86.0**

Table 2. Impact of different rank scaling functions $h(r)$. We compare the accuracy (%) of TaskIT using different functions against standard LoRA with fixed rank $r = 8$.

strict importance transfer to learned tasks whose similarity exceeds 0.35, ensuring that TaskIT primarily leverages sources that yield positive transfer.

C.2. Impact of Normalized Rank Contribution $h(r)$

Settings. We choose $h(r)$ by studying how accuracy scales with the LoRA rank r. First, we vary the LoRA rank $r \in \{1, \dots, 64\}$ and measure the relative accuracy gain over standard LoRA with rank $r = 1$ on six representative tasks from § 5.2.1. Second, we compare five candidate forms of $h(r)$, *i.e.* Uniform (1), Linear (r), Logarithmic ($\ln r$), Square Root ($\sqrt{r}$), and Root-Logarithmic ($\sqrt{\ln r}$), under a fixed memory budget, and report average accuracy in Tab. 2 against standard LoRA with fixed rank $r = 8$.

Results. Fig. 3 shows that accuracy grows sub-linearly with rank. Gains from increasing r are large at small ranks but become marginal from $r = 32$ to 64 despite doubling memory. Harder tasks (*e.g.* ImageNet) benefit more from higher ranks, whereas easier ones (*e.g.* Caltech101) quickly saturate. These trends motivate a concave $h(r)$ that rewards low ranks and penalizes linear memory growth at high ranks.

Tab. 2 confirms this observation. Uniform scaling yields the lowest average accuracy (80.3%) due to under-utilized capacity, while Linear scaling (83.0%) over-allocates budget to high ranks and overfits on simpler tasks. Logarithmic and Square Root scaling improve accuracy to 85.2% and 85.8%, respectively. Root-Logarithmic scaling ($h(r) \propto \sqrt{\ln r}$) achieves the best result at 86.0%, matching the empirically concave trend and balancing capacity with efficiency. Thus, we adopt $h(r) \propto \sqrt{\ln r}$ in all experiments.

Method	Backbone	Params (10^6)	Avg Mem. (GB)	Image Avg Acc.	NLP Avg Acc.	VL Avg Acc.
AdapterFusion [22]	OFA-b	0.9	1.6	58.6	76.2	67.3
	OFA-l	3.7	3.2	61.0	77.2	70.5
UniPT [5]	OFA-b	19.4	1.1	71.2	83.4	74.1
	OFA-l	49.2	2.8	73.6	84.7	76.0
AutoLoRA [31]	OFA-b	3.2	1.8	73.8	85.7	74.4
	OFA-l	8.8	4.5	**76.1**	**86.5**	**78.5**
TaskIT	OFA-b	1.9	0.8	73.9	84.6	75.3
	OFA-l	5.1	2.3	74.9	86.1	78.1

Table 3. Performance with other backbones. Only the most accurate Zero-FT, non-LoRA, and LoRA-based baselines in the cross-modal setting are included. Best results are **bolded**.

C.3. Impact of Other Backbones

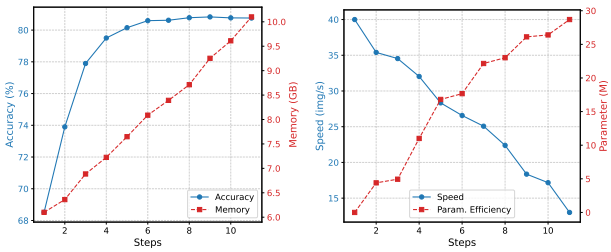
Settings. To assess architectural generalization, we extend TaskIT beyond Janus-Pro and adopt OFA [25] as the vision-language backbone following [26], using both Base (OFA-b, 180M) and Large (OFA-l, 470M) variants. We compare TaskIT with three SOTA baselines from § 5.2.1: AdapterFusion [22] (Zero-FT), UniPT [5] (Non-LoRA), and AutoLoRA [31] (LoRA-based). For each backbone, we report average accuracy over the cross-modal task suite in § 5.2.1, covering Image, NLP, and VL modalities, under the same memory budget.

Results. As shown in Tab. 3, TaskIT offers the best trade-off between memory and accuracy across all backbones. Compared with AutoLoRA, TaskIT achieves similar accuracy (within 0.4% on OFA-l) while reducing memory from 4.5 GB to 2.3 GB, a 48.8% saving. AdapterFusion uses fewer parameters but suffers from a lower accuracy. Although UniPT is memory-efficient on OFA-b, it scales poorly on OFA-l, requiring 49.2M trainable parameters versus only 5.1M for TaskIT.

C.4. Performance with Cold-start Setting

Settings. To assess the robustness of TaskIT in a resource-scarce cold-start setting, we initialize the multi-LoRA model with only a single set of modules trained on ImageNet-21k, and evaluate fine-tuning on the 19 downstream tasks of VTAB-1K [29] using a frozen ViT-Base backbone (86M parameters).

Results. Tab. 4 presents the results on VTAB-1K. Even with importance transfer limited to a single learned task (ImageNet-21k), TaskIT remains highly effective. With a memory budget factor $M_B = 0.6$, our method attains 79.3% average accuracy, outperforming the Zero-FT baseline AdapterFusion (59.1%) by over 20% and highlighting the limitations of simple upstream module fusion, particularly under cold-start with few learned tasks. Moreover, at $M_B = 0.6$, TaskIT attains a higher average ac-



(a) (b)

Figure 4. Impact of the memory budget M_B, showing the trade-offs between (a) memory usage (GB) and accuracy (%), and (b) training speed (img/sec) and trainable parameters (M).

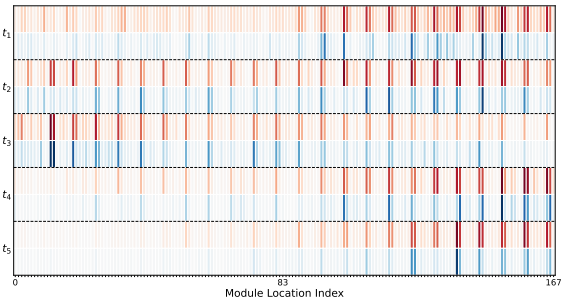


Figure 5. Predicted vs. Actual Importance of LoRA Modules. The heatmap compares the predicted importance (blue) with the actual importance (red) for LoRA modules across different location indices (x-axis) and tasks t_1 to t_5 (y-axis).

curacy (79.3% vs. 78.5% for UniPT) with 56% fewer trainable parameters. Compared with AutoLoRA, it nearly matches the state-of-the-art accuracy (79.5%) while using about 35% fewer parameters. This shows that our memory-aware scheduler effectively exploits the structural importance of the ImageNet module to prioritize critical parameters and maximize performance in the resource-constrained cold-start phase.

C.5. Impact of Memory Budget M_B

Settings. This experiment tests how the memory budget M_B affects TaskIT by varying M_B from a no-update setting to full LoRA fine-tuning, spanning 6.4 GB to 10.3 GB in 11 discrete steps. Following the cross-modal setup in § 5.2.1, we report the average fine-tuning accuracy and the number of trainable parameters at each memory level. The smallest budget corresponds to reusing existing LoRAs without inserting new modules, while the largest budget matches full LoRA fine-tuning.

Results. As shown in Fig. 4a, accuracy increases rapidly with M_B and saturates around step 6. Larger budgets allow TaskIT to insert more modules and assign higher ranks to important ones. Once these critical modules are fully activated, allocating memory to less important modules yields diminishing returns and may even cause mild overfitting. Fig. 4b further shows that the number of trainable parameters grows with M_B, which in turn slows training due to the additional gradient computations.

C.6. Effectiveness of Importance Prediction

Settings. We evaluate the effectiveness of the task importance predictor by comparing the predicted importance of each module with its actual importance across locations. We consider five tasks: t_1 Text-to-Image (CUB-200 [24]), t_2 Image Captioning (Flickr8k [8]), t_3 Image Classification (Pascal VOC [6]), t_4 Question Answering (SQuAD [23]), and t_5 Text Summarization (CNN/Daily Mail [7]). Following [17], we approximate the actual importance of each module by measuring the accuracy gain from individually inserting and fine-tuning that module. Fig. 5 visualizes the predicted (blue) and actual (red) importance across locations. We further quantify their alignment using Spearman's rank correlation coefficient ρ and corresponding p-values.

Results. In Fig. 5, the predicted importance closely tracks the actual importance at each location. Spearman's ρ confirms strong positive correlations: $\rho = 0.8862$ for t_2, 0.8837 for t_3, 0.8766 for t_4, and 0.8679 for t_5. Even for the challenging cross-modal task t_1, we obtain $\rho = 0.6819$. All correlations are statistically significant with $p < 10^{-24}$, indicating that TaskIT accurately predicts the relative importance of unknown modules.

We also observe task-specific patterns. For t_4 and t_5, importance concentrates in deeper locations, consistent with their need for high-level semantic reasoning and generation. For t_3, importance peaks in shallow and middle locations, where visual features are extracted and mapped to new classes. For t_2, importance spreads across most locations, reflecting its cross-modal nature and the need to adapt both early visual processing and later text generation stages.

C.7. Effectiveness of Module Memory Prediction

Settings. We evaluate the accuracy of our block-based memory predictor (§ 4.3) by comparing its predictions to ground-truth estimates from PyTorch profiling tools [4] and the linear index baseline [17], using the Janus-Pro backbone. Fig. 6 shows the normalized memory cost profiles, comparing the linear index prediction (gray), actual estimated memory (green), and our TaskIT prediction (blue).

Results. As shown in Fig. 6, TaskIT's predictions closely match the actual memory usage, achieving a Pearson correlation coefficient of $r = 0.94$. In contrast, the linear index prediction fails to capture the structural complexities of the Transformer architecture. We observe that memory usage within each block follows a sawtooth pattern, where modules like W_k, W_v, and W^2_{FFN} incurring higher costs due to

	Param(10^6)	Natural							Specialized				Structured								Average			
		Cifar100	Caltech101	DTD	Flower102	Pets	SVHN	Sun397	Camelyon	EuroSAT	Resisc45	Retinopathy	Clevr-Count	Clevr-Dist	DMLab	KITTI-Dist	dSpr-Loc	dSpr-Ori	sNORB-Azim	sNORB-Ele	Avg Natural	Avg Specialized	Avg Structured	Average
Full[12, 14]	85.8	68.9	87.7	64.3	97.2	86.9	87.4	38.8	79.7	95.7	84.2	73.9	56.3	58.6	41.7	65.5	57.5	46.7	25.7	29.1	75.9	83.4	47.6	68.9
Linear[12, 14]	0	64.4	85.0	63.2	97.0	86.3	36.6	51.0	78.5	87.5	68.5	74.0	34.3	30.6	33.2	55.4	12.5	20.0	9.6	19.2	69.1	77.1	26.9	57.6
BitFit[15, 28]	0.10	72.8	87.0	59.2	97.5	85.3	59.9	51.4	78.7	91.6	72.9	69.8	61.5	55.6	32.4	55.9	66.6	40.0	15.7	25.1	73.3	78.3	44.1	65.2
VPT[12, 14]	0.53	78.8	90.8	65.8	98.0	88.3	78.1	49.6	81.8	96.1	83.4	68.4	68.5	60.0	46.5	72.8	73.6	47.9	32.9	37.8	78.5	82.4	55.0	72.0
RS-Bypass. [13]	0.42	64.5	88.8	73.2	99.4	90.6	63.5	57.2	85.5	95.2	82.4	75.2	70.4	61.0	40.2	66.8	79.2	52.6	26.0	49.3	76.7	84.6	55.7	72.3
Adapter [9, 14]	0.16	69.2	90.1	68.0	98.8	89.9	82.8	54.3	84.0	94.9	81.9	75.5	80.9	65.3	48.6	78.3	74.8	48.5	29.9	41.6	79.0	84.1	58.5	73.9
LoRA[10]	0.29	67.1	91.4	69.4	98.8	90.4	85.3	54.0	84.9	95.3	84.4	73.6	82.9	**69.2**	49.8	78.5	75.7	47.1	31.0	44.0	79.5	84.6	59.8	74.5
AdaptFormer[2, 14]	0.16	70.8	91.2	70.5	99.1	90.9	86.6	54.8	83.0	95.8	84.4	76.3	81.9	64.3	49.3	80.3	76.3	45.7	31.7	41.1	80.6	84.9	58.8	74.7
NOAH [32]	0.36	69.6	92.7	70.2	99.1	90.4	86.1	53.7	84.4	95.4	83.9	75.8	82.8	68.9	49.9	81.7	81.8	48.3	32.8	44.2	80.3	84.9	61.3	75.5
FacT-TK [15]	0.06	70.6	90.6	70.8	99.1	90.7	88.6	54.1	84.8	96.2	84.5	75.7	82.6	68.2	49.8	80.7	80.8	47.4	33.2	43.0	80.6	85.3	60.7	75.6
RS [13]	0.55	75.2	92.7	71.9	99.3	**91.9**	86.7	**58.5**	86.7	95.6	85.0	74.6	80.2	63.6	50.6	80.2	85.4	55.7	31.9	42.0	82.3	85.5	61.2	76.3
Convpass [14]	0.33	72.3	91.2	72.2	99.2	90.9	**91.3**	54.9	84.2	96.1	85.3	75.6	82.3	67.9	51.3	80.0	85.9	53.1	36.4	44.4	81.7	85.3	62.7	76.6
HST [18]	0.78	76.7	**94.1**	74.8	99.6	91.1	91.2	52.3	87.1	96.3	**88.6**	76.5	**85.4**	63.7	**52.9**	81.7	87.2	56.8	35.8	**52.1**	82.8	87.1	64.5	78.1
LoSA [20]	0.19	82.5	92.8	76.1	99.7	90.5	82.0	55.8	86.6	97.1	87.0	76.7	81.5	62.3	48.6	82.1	94.2	61.7	47.9	45.6	82.8	86.9	65.5	78.4
AdapterFusion [22]	0.23	67.4	67.8	55.5	77.9	73.3	58.3	41.3	69.8	85.9	70.7	51.9	55.1	50.8	35.0	63.8	59.0	36.2	26.0	31.4	63.1	69.6	44.7	59.1
UniPT [5]	1.21	82.7	93.0	76.2	99.7	90.6	82.1	55.9	86.8	97.1	87.2	77.0	81.8	62.4	49.0	82.2	94.3	61.8	48.0	45.8	82.9	87.0	65.6	78.5
AutoLoRA [31]	0.82	**83.7**	93.7	**77.7**	**99.8**	91.5	82.9	56.7	**87.6**	**97.7**	88.4	**77.6**	82.2	64.2	50.6	**84.0**	**94.9**	**63.2**	**50.2**	48.5	**83.7**	**87.8**	**67.2**	**79.5**
TaskIT, $M_B = 0.4$	0.37	82.6	92.7	76.7	99.7	90.7	81.4	55.5	86.8	97.1	87.4	76.7	80.4	62.1	48.8	82.4	92.7	61.4	48.7	46.7	82.8	87.0	65.3	78.3
TaskIT, $M_B = 0.5$	0.44	83.1	93.0	77.1	99.7	91.0	82.0	56.0	87.1	97.3	87.8	77.1	81.2	63.0	49.5	83.0	93.6	62.2	49.3	47.6	83.1	87.3	66.2	78.9
TaskIT, $M_B = 0.6$	0.53	83.5	93.5	77.5	**99.8**	91.4	82.8	56.5	87.5	97.6	88.2	77.5	81.8	63.8	50.2	83.5	94.5	62.8	49.8	48.0	83.6	87.7	66.8	79.3

Table 4. Comparison to state-of-the-art parameter-efficient finetuning methods on VTAB-1K [29]. Following standard practice, the final "Average" is the average of three preceding groupwise averages. Parameters denotes the number of learnable parameters excluding the final classification layer. **Best results** are bolded, and second-best underlined.

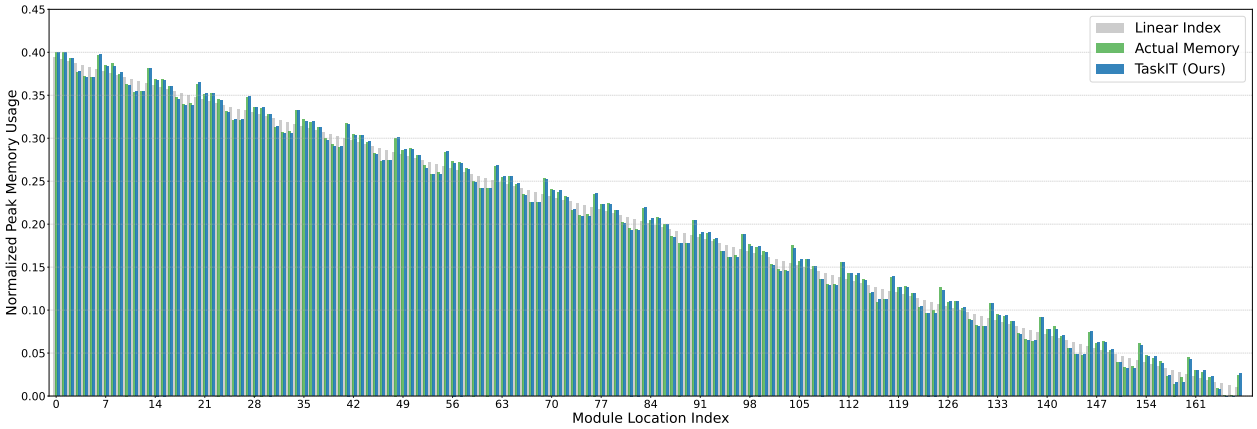


Figure 6. Comparison of normalized module memory usage predictions/estimations. TaskIT accurately captures the structural sawtooth memory pattern of Transformer blocks, aligning closely with the actual memory usage compared to the linear index baseline.

the retention of large activations, while W_o and W^1_{FFN} are more memory-efficient, relying on smaller activations.

C.8. Edge-device evaluation and cumulative overhead

We evaluate TaskIT on a Jetson Orin Nano (8GB GPU) under a QLoRA setting, and show the edge-device platform in Fig. 7. Janus-Pro-1B is loaded in 8-bit via *BitsAndBytes*, while fine-tuning is performed in 16-bit for 10 epochs. We report accuracy, peak memory, training latency, and energy consumption in Table 5, and summarize the cumulative runtime and memory overhead of the major TaskIT components in Table 6. The results show that TaskIT preserves the best memory–accuracy trade-off under quantization, while remaining practical on real edge hardware, and that its additional system overhead remains modest relative to sparse

Figure 7. Jetson Orin Nano platform used for the edge-device evaluation.

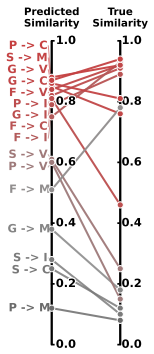


Figure 8. Validation of the task-similarity approximation by comparing the predicted similarities against the true similarities after training the new-task LoRA modules.

fine-tuning.

C.9. Task similarity approximation and sensitivity analysis

Eq. (4) can be interpreted as a first-order Taylor approximation of the representation update induced by adaptation. To validate the approximation empirically, we train the new-task LoRA modules w_{n+1} and compute the true similarity using $x(\theta + w_i; d)$ and $x(\theta + w_{n+1}; d)$ for all 16 learned→new task pairs in the setting of Appendix C.1 in Fig. 8. The predicted similarities closely match the true similarities, with Pearson correlation $r = 0.86$.

With batch-mean loss reduction and cosine similarity dominated by direction rather than magnitude, the method is not sensitive to the exact step size η. In practice, we set $\eta = 10$ so that the perturbation magnitude satisfies $\|\eta \nabla_x L\|/\|x\| \approx 0.1$. We further evaluate the sensitivity to both η and the task-similarity threshold across four cross-modal tasks, and observe stable behavior over a broad hyperparameter range in Fig. 9.

C.10. Module importance prediction

Fig. 10 validates the predicted module importance against realized performance gains after insertion. The formulation in Eq. (3) is evaluated only on shared backbone projection matrices W_k, making it agnostic to modality-specific encoders, decoders, and task heads. We additionally provide calibration plots on VQA, ImageNet, and MNLI by correlating the predicted module importance with the ablated accuracy drop, which we treat as the true importance.

We also ablate Eq. (3) by removing the denominator $|x(W_k; d)|_1$ in Table 7, replacing new-task data with learned-task data, and substituting the ℓ_1 norm with ℓ_2. The ratio-based formulation used in TaskIT remains the most robust among these alternatives.

C.11. Module insertion scheduler and additional analyses

We compare the proposed dynamic-programming scheduler with greedy search and evolutionary search in Table 8. The DP scheduler achieves the best downstream accuracy while keeping search overhead low. This supports the claim that a more structured search is beneficial even under tight memory constraints.

The DP formulation assumes that the incremental memory cost ΔM depends only on the nearest predecessor k_s. This is exact when k_s and the candidate location lie on the same sequential backward path, including across different blocks and within the same FFN. When they belong to parallel $Q/K/V$ branches of the same attention block, the approximation can over-count at most one branch activation, and the error does not propagate across blocks.

For scalability with the number of learned tasks, the ideal feature of the new task is obtained with one backward pass, and similarities to prior tasks are then computed with n forward passes. Because the learned tasks are processed sequentially, peak memory remains constant, while latency scales linearly with the number of prior tasks in Fig. 11.

We also report explicit accuracy–memory trade-offs of TaskIT at larger budgets in Table 9. Under 10.5GB and 14.5GB budgets, TaskIT continues to improve average accuracy as more memory is made available, showing that the framework supports flexible budget-aware adaptation.

References

[1] David Chen and William B Dolan. Collecting highly parallel data for paraphrase evaluation. In *Proceedings of the ACL*, pages 190–200, 2011. 4

[2] Shoufa Chen, Chongjian Ge, Zhan Tong, Jiangliu Wang, Yibing Song, Jue Wang, and Ping Luo. Adaptformer: Adapting vision transformers for scalable visual recognition. *Advances in Neural Information Processing Systems*, 35:16664–16678, 2022. 3, 7

[3] Alexandra Chronopoulou, Matthew E Peters, Alexander Fraser, and Jesse Dodge. Adaptersoup: Weight averaging to improve generalization of pretrained language models. *arXiv preprint arXiv:2302.07027*, 2023. 2

Method	CIFAR-100 (VTAB-1K)				MRPC			
	Acc. (%) ↑	Mem. (GB) ↓	Lat. (min) ↓	Eng. (kJ) ↓	Acc. (%) ↑	Mem. (GB) ↓	Lat. (min) ↓	Eng. (kJ) ↓
LoRAHub	68.2	**5.2**	**11.5**	**9.6**	78.5	**4.7**	**22.3**	**18.9**
LoRA	76.2	8.9*	112.4	94.8	87.1	8.1*	205.5	180.8
UniPT	75.1	7.7	63.5	51.3	86.8	7.3	147.4	131.6
TaskIT (Ours)	**76.4**	6.5	65.7	52.4	**87.5**	5.8	143.2	125.4

Table 5. Edge-device evaluation on Jetson Orin Nano under a QLoRA setup. * indicates that memory swapping occurred.

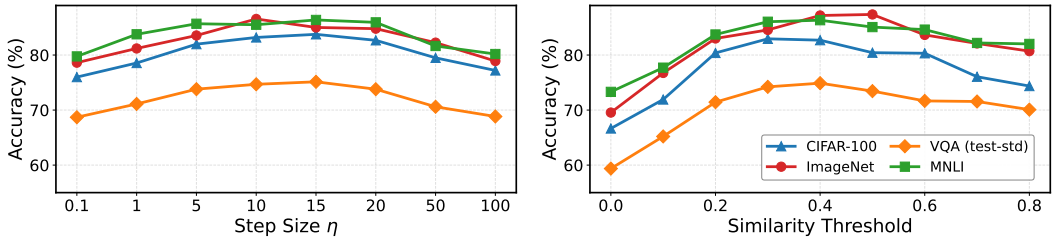


Figure 9. Sensitivity analysis with respect to the simulation step size η and the task-similarity threshold across four cross-modal tasks.

TaskIT module	Time (s)	Ratio (%)	Peak Mem. (GB)
Importance estimation	53.7	1.4	4.9
Memory profiling	92.2	2.3	4.6
DP scheduler	28.5	0.7	0.04
Sparse fine-tuning	3768.2	95.6	6.5

Table 6. Cumulative overhead of the major components of TaskIT on CIFAR-100 on Jetson. The extra overhead is modest relative to sparse fine-tuning.

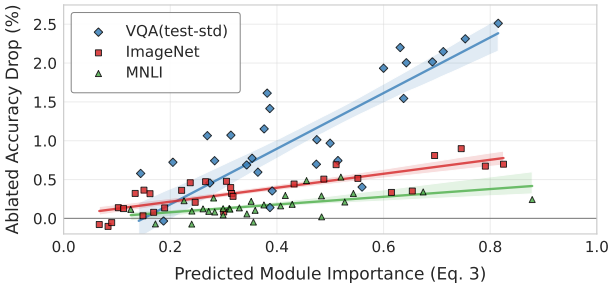


Figure 10. Calibration plots correlating predicted module importance with ablated accuracy drop on VQA, ImageNet, and MNLI.

Eq. (3) variant	Correlation ↑	Acc. (%) ↑
w/o denominator (no ratio)	0.39	80.5
w/ learned-task data	0.53	81.2
w/ ℓ_2 norm	0.68	83.3
TaskIT (ours)	**0.71**	**83.6**
Ablated accuracy drop	1.00	85.5

Table 7. Ablation of the importance metric in Eq. (3). The last row is the oracle importance measured by ablated accuracy drop after full-LoRA fine-tuning on the new task.

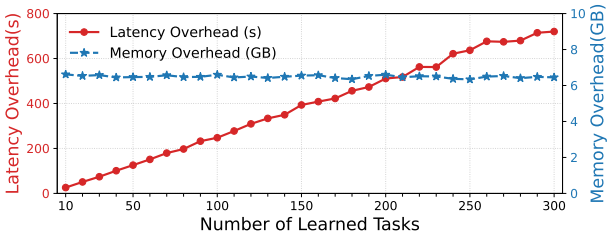


Figure 11. Scalability of task-similarity computation with respect to the number of learned tasks. Peak memory remains constant, while latency scales approximately linearly.

[4] PyTorch Developers. Pytorch profiler. https://docs.pytorch.org/stable/profiler.html. 6

[5] Haiwen Diao, Bo Wan, Ying Zhang, Xu Jia, Huchuan Lu, and Long Chen. Unipt: Universal parallel tuning for transfer learning with efficient parameter and memory. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 28729–28740, 2024. 3, 5, 7

[6] Mark Everingham, Luc Van Gool, Christopher KI Williams, John Winn, and Andrew Zisserman. The pascal visual object classes (voc) challenge. *International journal of computer vision*, 88(2):303–338, 2010. 6

[7] Karl Moritz Hermann, Tomas Kocisky, Edward Grefenstette, Lasse Espeholt, Will Kay, Mustafa Suleyman, and Phil Blunsom. Teaching machines to read and comprehend. *Advances in neural information processing systems*, 28, 2015. 4, 6

[8] Micah Hodosh, Peter Young, and Julia Hockenmaier. Framing image description as a ranking task: Data,

Method	Search overhead		Sparse LoRA tuning		
	Time (s) ↓	Memory (MB) ↓	Acc. (%) ↑	Param (M) ↓	Memory (GB) ↓
Greedy search	**1.2**	**8.2**	77.5	**9.1**	7.9
Evolutionary search	628.5	26.7	80.1	15.4	8.0
DP search (ours)	13.2	42.3	**80.4**	16.8	**7.8**

Table 8. Comparison of the scheduler in TaskIT with greedy and evolutionary search.

Method	**Nat. Avg Acc. ↑**	**Spec. Avg Acc. ↑**	**Struc. Avg Acc. ↑**	**Memory (GB) ↓**
TaskIT ($M_B = 10.5$ GB)	84.1	87.6	68.5	**10.3**
TaskIT ($M_B = 14.5$ GB)	**84.7**	**87.7**	**70.2**	14.4

Table 9. Accuracy–memory trade-offs of TaskIT under larger memory budgets.

models and evaluation metrics. *Journal of Artificial Intelligence Research*, 47:853–899, 2013. 6

[9] Neil Houlsby, Andrei Giurgiu, Stanislaw Jastrzebski, Bruna Morrone, Quentin De Laroussilhe, Andrea Gesmundo, Mona Attariyan, and Sylvain Gelly. Parameter-efficient transfer learning for nlp. In *Proceedings of the ICML*, pages 2790–2799. PMLR, 2019. 3, 7

[10] Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, Weizhu Chen, et al. Lora: Low-rank adaptation of large language models. *ICLR*, 1(2):3, 2022. 3, 7

[11] Chengsong Huang, Qian Liu, Bill Yuchen Lin, Tianyu Pang, Chao Du, and Min Lin. Lorahub: Efficient cross-task generalization via dynamic lora composition. In *Proceedings of the COLM*, 2024. 2

[12] Menglin Jia, Luming Tang, Bor-Chun Chen, Claire Cardie, Serge Belongie, Bharath Hariharan, and Ser-Nam Lim. Visual prompt tuning. In *European conference on computer vision*, pages 709–727. Springer, 2022. 2, 7

[13] Zeyinzi Jiang, Chaojie Mao, Ziyuan Huang, Ao Ma, Yiliang Lv, Yujun Shen, Deli Zhao, and Jingren Zhou. Res-tuning: A flexible and efficient tuning paradigm via unbinding tuner from backbone. *Advances in Neural Information Processing Systems*, 36:42689–42716, 2023. 2, 3, 7

[14] Shibo Jie and Zhi-Hong Deng. Convolutional bypasses are better vision transformer adapters. *arXiv preprint arXiv:2207.07039*, 2022. 2, 3, 7

[15] Shibo Jie and Zhi-Hong Deng. Fact: Factor-tuning for lightweight adaptation on vision transformer. In *Proceedings of the AAAI conference on artificial intelligence*, pages 1060–1068, 2023. 3, 7

[16] Chris Dongjoo Kim, Byeongchang Kim, Hyunmin Lee, and Gunhee Kim. Audiocaps: Generating captions for audios in the wild. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 119–132, 2019. 4

[17] Ji Lin, Ligeng Zhu, Wei-Ming Chen, Wei-Chen Wang, Chuang Gan, and Song Han. On-device training under 256kb memory. *Advances in Neural Information Processing Systems*, 35:22941–22954, 2022. 6

[18] Weifeng Lin, Ziheng Wu, Wentao Yang, Mingxin Huang, Jun Huang, and Lianwen Jin. Hierarchical side-tuning for vision transformers. *arXiv preprint arXiv:2310.05393*, 2023. 3, 7

[19] Xinhao Mei, Chutong Meng, Haohe Liu, Qiuqiang Kong, Tom Ko, Chengqi Zhao, Mark D Plumbley, Yuexian Zou, and Wenwu Wang. Wavcaps: A chatgpt-assisted weakly-labelled audio captioning dataset for audio-language multimodal research. *arXiv preprint arXiv:2303.17395*, 2023. 4

[20] Otniel-Bogdan Mercea, Alexey Gritsenko, Cordelia Schmid, and Anurag Arnab. Time-memory-and parameter-efficient visual adaptation. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 5536–5545, 2024. 2, 3, 4, 7

[21] Rui Pan, Xiang Liu, Shizhe Diao, Renjie Pi, Jipeng Zhang, Chi Han, and Tong Zhang. Lisa: Layerwise importance sampling for memory-efficient large language model fine-tuning. *Advances in Neural Information Processing Systems*, 37:57018–57049, 2024. 3

[22] Jonas Pfeiffer, Aishwarya Kamath, Andreas Rücklé, Kyunghyun Cho, and Iryna Gurevych. Adapterfusion: Non-destructive task composition for transfer learning. 2021. 2, 5, 7

[23] Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. Squad: 100,000+ questions for machine comprehension of text. *arXiv preprint arXiv:1606.05250*, 2016. 6

[24] Catherine Wah, Steve Branson, Peter Welinder, Pietro Perona, and Serge Belongie. The caltech-ucsd birds-200-2011 dataset. 2011. 6

[25] Peng Wang, An Yang, Rui Men, Junyang Lin, Shuai Bai, Zhikang Li, Jianxin Ma, Chang Zhou, Jingren Zhou, and Hongxia Yang. Ofa: Unifying architectures, tasks, and modalities through a simple sequence-to-sequence learning framework. In *International conference on machine learning*, pages 23318–23340. PMLR, 2022. 5

[26] Chengyue Wu, Teng Wang, Yixiao Ge, Zeyu Lu, Ruisong Zhou, Ying Shan, and Ping Luo. π-tuning: Transferring multimodal foundation models with optimal multi-task interpolation. In *Proceedings of the ICML*, 2023. 3, 4, 5

[27] Dejing Xu, Zhou Zhao, Jun Xiao, Fei Wu, Hanwang Zhang, Xiangnan He, and Yueting Zhuang. Video question answering via gradually refined attention over appearance and motion. In *Proceedings of the 25th ACM international conference on Multimedia*, pages 1645–1653, 2017. 4

[28] Elad Ben Zaken, Shauli Ravfogel, and Yoav Goldberg. Bitfit: Simple parameter-efficient fine-tuning for transformer-based masked language-models. In *Proceedings of the ACL*, 2022. 2, 7

[29] Xiaohua Zhai, Joan Puigcerver, Alexander Kolesnikov, Pierre Ruyssen, Carlos Riquelme, Mario Lucic, Josip Djolonga, Andre Susano Pinto, Maxim Neumann, Alexey Dosovitskiy, et al. A large-scale study of representation learning with the visual task adaptation benchmark. *arXiv preprint arXiv:1910.04867*, 2019. 5, 7

[30] Qingru Zhang, Minshuo Chen, Alexander Bukharin, Nikos Karampatziakis, Pengcheng He, Yu Cheng, Weizhu Chen, and Tuo Zhao. Adalora: Adaptive budget allocation for parameter-efficient fine-tuning. In *Proceedings of the ICLR*, 2023. 3

[31] Ruiyi Zhang, Rushi Qiang, Sai Ashish Somayajula, and Pengtao Xie. Autolora: Automatically tuning matrix ranks in low-rank adaptation based on meta learning. In *Proceedings of the ACL*, 2024. 1, 3, 4, 5, 7

[32] Yuanhan Zhang, Kaiyang Zhou, and Ziwei Liu. Neural prompt search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2024. 3, 7