A Related works

Pre-trained language models. PLMs, which are trained on extensive datasets for a common task such as predicting masked words [1, 2, 3, 26, 55, 56, 57, 58] or the next word [59, 60] in a sentence, play a vital role in facilitating knowledge transfer to downstream tasks. They have demonstrated remarkable achievements across various applications, consistently delivering state-of-the-art outcomes. Furthermore, scaling up PLMs has proven to yield predictable enhancements in performance for these downstream tasks [61, 62]. Consequently, the size of released PLMs has progressively grown, reaching an unprecedented scale of 100 billion parameters [9, 10, 11, 60, 63, 64]. Such large-scale PLMs unveil extraordinary capabilities, enabling zero-shot or in-context learning [59, 60] for a broad spectrum of tasks. Nevertheless, transfer learning remains a prevalent approach for effectively deploying these models in new task scenarios [29, 65, 66], which post unparalleled requirements on the computing resources.

Parameter-efficient fine-tuning. With the advent of large-scale PLMs, a new method that aims to reduce storage requirements, PEFT, has been proposed [14, 15, 19]. PEFT adds and trains a small number of parameters while matching the performance of full fine-tuning. There are various ways to add new parameters. For example, Houlsby et al. [14] and Pfeiffer et al. [16] insert small bottleneck modules (adapters) to the PLM. LoRA [17] injects rank decomposition matrices into the pre-trained weights. HiWi [13] inserts the pre-trained parameters to a low-rank adapter. (IA)^3 [29] scales the pre-trained weight with a trainable vector. Prompt-based methods [15, 19] append a sequence of trainable vectors to the word embeddings or attention components. Recently, some unified methods, which combine multiple PEFT methods in a heuristic way [18] or with the technique of neural architecture search [49, 67, 68], have also been proposed. Though PEFTs save the storage by a large margin compared to full fine-tuning, they still require a similar memory footprint during training as full fine-tuning [20, 21] because of the activation memory.

Memory-efficient training. Memory-efficient training aims to reduce the memory footprint during the training process. Reversible neural networks [40, 41, 42] reduce the activation memory by recomputing the activations with the outputs during back-propagation. Gradient checkpointing [69] trade computation for memory by dropping some intermediate activations and recovering them from an extra forward pass. The activation memory is $\mathcal{O}(1)$ and $\mathcal{O}(\sqrt{N})$ for reversible neural networks and gradient checkpointing, respectively. MEFT is the first method that is proposed to modify a PLM to its reversible variant. When applying MEFT on a deeper model, one can use gradient checkpointing to further reduce the activation memory for the layers with vanilla gradient.

Network compressions, like pruning [70, 71] and knowledge distillation [22, 23, 72], save the memory footprint for both training and inference. They compress a PLM to a smaller model by either deleting unimportant parameters or distilling knowledge from the PLM to the smaller model. Treating a PLM as a feature extractor and avoiding its gradient calculation is also an effective way to reduce the activation memory [20, 21]. However, these methods normally require extra pre-training before fine-tuning, or achieve a lower performance compared to full fine-tuning when using the same PLM.

B Limitations

We acknowledge the main limitation of this work is that we only evaluate our proposed methods on a limited amount of tasks and don't conduct experiments on the encoder-decoder models. The main reason for the limited amount of tasks is that our computing resources are constrained. In addition, the major criterion for our selection of the underlying models is that we could find many strong baselines on them without reproduction. BERT and RoBERTa fulfill this criterion very well on the GLUE benchmark. Regarding the encoder-decoder model, recently there is a clear trend of applying a decoder-only model on sequence-to-sequence tasks. Therefore, we apply OPT in this paper and plan to include LLAMA [11] for the instruction-finetuning data in the future.

Another limitation of MEFT is its lower score when trained in FP16 and on a deeper model. We have discussed this problem in §4.2. In sum, more reconstruction error is introduced by FP16 due to its numerical range and by a deeper model because of the error accumulation. Fortunately, the results are still comparable to the PEFT baselines when trained in FP16. Even trained in FP32, the activation memory footprints don't increase compared to FP16. One only needs to spend more training time in FP32 when using the same batch size as in FP16 (about 20% more training time). However, since

MEFTs reduce the memory footprint, a larger batch size during training is possible, which can save some training time. For deeper models, we offer a practical and effective setting in Figure 7.

Last but not least, when fine-tuning larger models, like OPT$_{1.3B}$ and OPT$_{6.7B}$ [9], the peak memory footprint is occupied by the model parameters rather than the activation (see Table 3). One needs to combine other techniques with MEFT to reduce the peak memory footprint, like loading the model in FP16 or even in int8 rather than in FP32, combining MEFT with ZeRO [73] as in Table 6.

C Step-by-step design for MEFT$_1$

For the reader's easy understanding, in this section, we explain MEFT$_1$ step-by-step. First, let's re-emphasize the guiding principles for our design: (1) For each reversible layer, we must have two inputs and two outputs as in Figure 3a. (2) We need to follow the starting point hypothesis. I.e. whenever we modify a PLM layer, we need to ensure the modified layer has almost the same output as the original PLM layer if we input the same input of the original PLM layer to the modified layer at the beginning of training. If the outputs are not similar, they become even more dissimilar after multiple layers, tearing down the PLM's initialization.

As shown in Figure 8a, for the first PLM layer, $\boldsymbol{h}_0$ is the input and $\boldsymbol{h}_1$ is the output. In Figure 8b, the inputs to the first reversible layer is $\boldsymbol{h}_0^1 = \boldsymbol{h}_0^2 = \boldsymbol{h}_0$. Recapping the architecture of $\mathcal{F}_1$ in Figure 4c (up), we simply insert an adapter in parallel to the two consecutive feed-forward layers, and initialize the adapter as $W_{down}, W_{up} \sim \mathcal{N}(0, 0.02^2)$, which results in $\boldsymbol{h}_1 \approx \mathcal{F}_1(\boldsymbol{h}_0^2)$ since $\boldsymbol{h}_0^2 = \boldsymbol{h}_0$. If we set $\lambda \to 0$, $\boldsymbol{h}_1^1 = \lambda \boldsymbol{h}_0^1 + \mathcal{F}_1(\boldsymbol{h}_0^2) \approx \boldsymbol{h}_1$. In this way, $\boldsymbol{h}_1^1$ plays the role of preserving the starting point. Now let's consider $\boldsymbol{h}_1^2$. Due to our initialization of the adapter, the output from $\mathcal{G}_1$ ($\mathcal{G}_1$ is simply an adapter as in Figure 4c (down)) is close to $\mathbf{0}$. So $\boldsymbol{h}_1^2 = \beta \boldsymbol{h}_0^2 + \mathcal{G}_1(\boldsymbol{h}_1^1) \approx \beta \boldsymbol{h}_0 + \mathbf{0} = \beta \boldsymbol{h}_0$. After switching the order of $\boldsymbol{h}_1^1$ and $\boldsymbol{h}_1^2$, $\boldsymbol{h}_1^1 \approx \beta \boldsymbol{h}_0$ and $\boldsymbol{h}_1^2 \approx \boldsymbol{h}_1$.

For the second reversible layer, if we don't switch the order of $\boldsymbol{h}_1^1$ and $\boldsymbol{h}_1^2$, it looks like Figure 8c. The input to $\mathcal{F}_2$ is $\beta \boldsymbol{h}_0$, which breaks down the representation continuity of a PLM since the input to the pre-trained $\mathcal{F}_2$ should be close to $\boldsymbol{h}_1$. If we switch their order as in Figure 8d, we preserve the representation continuity. And it results in $\boldsymbol{h}_2^1 = \lambda \beta \boldsymbol{h}_0 + \mathcal{F}_2(\boldsymbol{h}_1) \approx \boldsymbol{h}_2$ due to $\lambda \to 0$ and $\boldsymbol{h}_2 \approx \mathcal{F}_2(\boldsymbol{h}_1)$. Similar to the first reversible layer, $\boldsymbol{h}_2^2 \approx \beta \boldsymbol{h}_1$. After switching, $\boldsymbol{h}_2^1 \approx \beta \boldsymbol{h}_1$ and $\boldsymbol{h}_2^2 \approx \boldsymbol{h}_2$. By analogy, for the n^{th} reversible layer, $\boldsymbol{h}_n^1 \approx \beta \boldsymbol{h}_{n-1}$ and $\boldsymbol{h}_n^2 \approx \boldsymbol{h}_n$.

After the final layer, we simply take the mean of two outputs as $\boldsymbol{h}'_N = (\boldsymbol{h}_N^1 + \boldsymbol{h}_N^2)/2$, and input $\boldsymbol{h}'_N$ to a task-specific head, like a classification layer. The design procedure is similar for MEFT$_2$ and MEFT$_3$. In sum, order switching is mainly for preserving the representation continuity, and setting the scaling factors close to 0 is mainly for preserving the starting point.

D Implementation details of the question-answering tasks

Compared to GLUE tasks where all tasks are classification tasks and the classification heads are randomly initialized, the question-answering tasks are sequence-to-sequence tasks and need the pre-trained output layer that shares the same parameters as the word embedding layer. The output

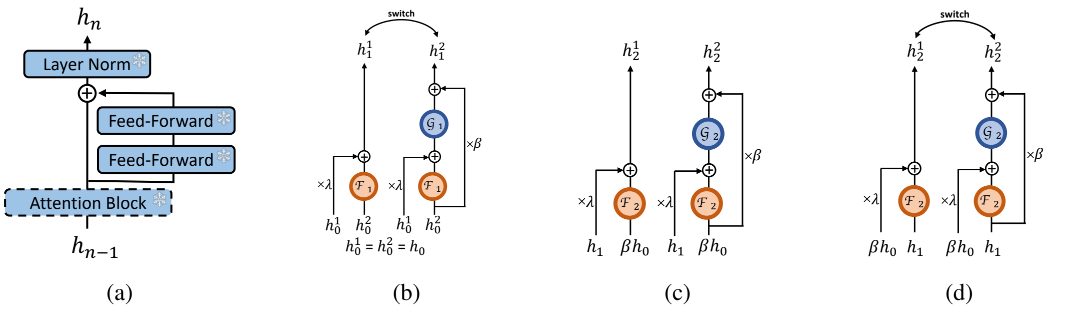
h_n
Layer Norm
Feed-Forward
Feed-Forward
Attention Block
h_{n-1}
(a)
switch
h_1^1 h_1^2
$\mathcal{G}_1$ $\times\beta$
$\times\lambda$ $\mathcal{F}_1$ $\times\lambda$ $\mathcal{F}_1$
h_0^1 h_0^2 h_0^1 h_0^2
$h_0^1 = h_0^2 = h_0$
(b)
h_2^1 h_2^2
$\mathcal{G}_2$ $\times\beta$
$\times\lambda$ $\mathcal{F}_2$ $\times\lambda$ $\mathcal{F}_2$
h_1 βh_0 h_1 βh_0
(c)
switch
h_2^1 h_2^2
$\mathcal{G}_2$ $\times\beta$
$\times\lambda$ $\mathcal{F}_2$ $\times\lambda$ $\mathcal{F}_2$
βh_0 h_1 βh_0 h_1
(d)

Figure 8: (a) The n^{th} PLM layer; (b) The first MEFT$_1$ layer; (c) The second MEFT$_1$ layer without order switching; (d) The second MEFT$_1$ layer.

layer requires the continuity of representation. I.e. at the beginning of training, the input to the output layer, $\boldsymbol{h}'_N$, should be close to $\boldsymbol{h}_N$. Therefore, we need to do a modification to $\boldsymbol{h}'_N$ instead of using $\boldsymbol{h}'_N = (\boldsymbol{h}^1_N + \boldsymbol{h}^2_N)/2$.

Here we introduce a new scaling factor γ and require $\gamma \to 0$. For MEFT$_1$, since $\boldsymbol{h}^2_N \approx \boldsymbol{h}_N$ (see Table 1), we set $\boldsymbol{h}'_N = \gamma \boldsymbol{h}^1_N + \boldsymbol{h}^2_N \approx \boldsymbol{h}^2_N \approx \boldsymbol{h}_N$. Similarly, $\boldsymbol{h}'_N = \boldsymbol{h}^1_N + \gamma \boldsymbol{h}^2_N \approx \boldsymbol{h}^1_N \approx \boldsymbol{h}_N$ for MEFT$_2$, and $\boldsymbol{h}'_N = \gamma \boldsymbol{h}^1_N + \boldsymbol{h}^2_N \approx \boldsymbol{h}^2_N \approx \boldsymbol{h}_N$ for MEFT$_3$. Without any tuning, we set $\gamma = 0.1$ as other tuned scaling factors by default.

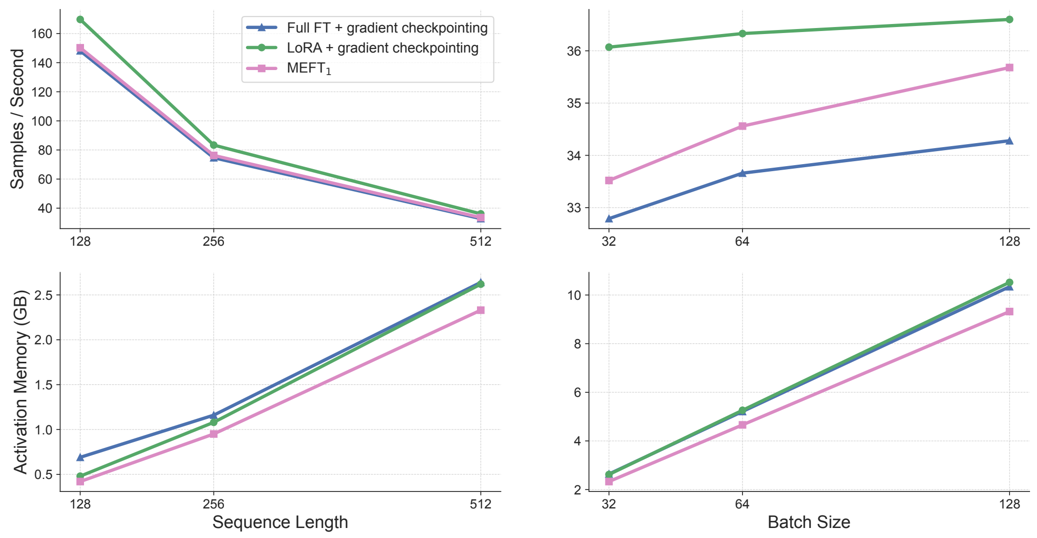

Figure 9: Throughput and activation memory for different sequence length and batch sizes on BERT$_{\text{base}}$. By default, the sequence length is 512 and the batch size is 32. For your reference, LoRA's throughput is 52.7 samples/second without gradient checkpointing for the default setting. Overall, MEFT shares the same level of throughput as LoRA with gradient checkpointing, while it is the lower bound of the activation memory for different settings.

E More results

E.1 Compared to gradient checkpointing

Previously, we only theoretically stated that the activation memory for reversible network and gradient checkpointing is $\mathcal{O}(1)$ and $\mathcal{O}(\sqrt{N})$, respectively. In addition, we didn't compare the training time of MEFT with PEFT in detail. Here we offer some empirical results for your better understanding.

In Figure 9, we compare activation memory and throughput among MEFT , LoRA with gradient checkpointing and Full FT with gradient checkpointing. The throughput for all three methods is at the same level, maximum 12% difference between LoRA and MEFT when the sequence length is 128 and the batch size is 32. With an increasing sequence length, the gap becomes narrower to 7.5%. Notably, the throughput for LoRA without gradient checkpointing is 52.7 samples/second. With gradient checkpointing, it is 36.1 samples/second, 69% of the original throughput. For MEFT with the same setting, it is 33.5 samples/second, 64% of LoRA's throughput without gradient checkpointing. In sum, MEFT's throughput is at the same level as LoRA's with gradient checkpointing, and about 64% of LoRA's without gradient checkpointing. In addition, MEFT's activation memory is always the lower bound among these three methods. The gap between LoRA with gradient checkpointing and MEFT becomes larger with an increasing sequence length and batch size.

E.2 Compared to quantization methods

Quantization is an orthogonal method to MEFT, which reduces the memory footprint of training or inference by reducing the parameter size to fewer bits and using low-bit-precision matrix multiplication. There are mainly three different quantization methods: (1) Post-training quantization

Table 5: Compared to QLoRA. $r = 8$ for all methods. Experimental setting stays the same as the default setting in Figure 9.

Method	**Activation Memory (GB)**	**Samples/Second**
LoRA + gradient checkpointing	2.62	36.1
QLoRA + gradient checkpointing	2.97	8.7
MEFT$_1$	2.33	33.5

Table 6: Combine MEFT with ZeRO.

Method	**Peak Memory (GB)**	**Activation Memory (GB)**
MEFT$_1$	28.2	8.2
MEFT$_1$ + ZeRO	6.4	6.4

[74, 75] that quantizes a trained model after pre-training or fine-tuning; (2) Lower-bit optimizer [76] that stores the optimizer state with lower precision and de-quantizes it only for the optimization, similarly to FP16 or BF16 mixed precision training but with lower-bit; (3) Lower-bit frozen LLM with LoRA, i.e. QLoRA [77], that applies 4-bit quantization to compress the LLM. During fine-tuning, QLoRA backpropagates gradients through the frozen 4-bit quantized LLM into the low-rank adapters. Notably, the computation data type for QLoRA is BF16. It de-quantizes weights to the computation data type to perform the forward and backward passes.

To some extent, all these three methods are orthogonal to our method and can be combined with MEFT: (1) Post-training quantization is mainly for reference and it can be applied to any trained models; (2) 8-bit Adam can also be applied to any models trained based on a gradient; (3) QLoRA is a combination of (1) and (2). For QLoRA, we conducted some experiments on BERT$_{\text{base}}$ with the default setting as Figure 9. As shown in Table 5, METF$_1$ saves the most activation memory while having a similar throughput as LoRA with gradient checkpointing. The reason for the larger activation memory of QLoRA than LoRA is that it has an additional de-quantization step, which also causes its smallest throughput.

E.3 Combine MEFT with ZeRO

ZeRO [73] saves memory by partitioning the model's parameters and optimizer state among GPUs or between GPU and CPU. This method is orthogonal to MEFT, since MEFT saves memory from activations. We conduct some experiments on OPT$_{1.3\text{B}}$ by combining our method with DeepSpeed [78] ZeRO stage 3 that offloading model's parameters and the optimizer state to CPUs. As shown in Table 6, ZeRO significantly reduces the memory footprint from the model's parameters, therefore reducing MEFT's peak memory from 28.2GB to 6.4GB.

Table 7: Fine-tuning settings. Check §4.2 for the fine-tuning setting on BART.

Hyper-parameter	**GLUE**		**Question-Answering**
	RTE, MRPC, STS-B, CoLA	**SST-2, QNLI, QQP, MNLI**	
Learning Rate	$\{5, 6, 7, 8\} \cdot 10^{-4}$	$\{3, 4, 5\} \cdot 10^{-4}$	$\{1, 3, 5, 7\} \cdot 10^{-4}$
Batch Size	$\{16, 32\}$	$\{16, 32\}$	$\{8, 16, 32\}$
Max Epochs	$\{20, 40\}$	$\{10, 20\}$	$\{3, 5, 10\}$
Weight Decay	0.1	0.1	0.1
Max Gradient Norm	1	1	1
Warmup Ratio	0.06	0.06	0.06
Learning Rate Decay	Linear	Linear	Linear

Table 8: Statics of datasets

Task	RTE	MRPC	STS-B	CoLA	SST-2	QNLI	QQP	MNLI-m	MNLI-mm
#Training	2.5k	3.7k	5.8k	8.6k	67.4k	104.7k	363.8k	392.7k	
#Development	0.3k	0.4k	1.5k	1k	0.9k	5.5k	40.4k	9.8k	9.8k
Task	OpenBookQA	PIQA	ARC-E	ARC-C	SciQ				
#Training	5.0k	16.1k	2.3k	1.1k	11.7k				
#Development	0.5k	3.1k	2.4k	1.2k	1k				

Table 9: Statics of models

Model	**#Parameter**	**#Layer**	$\mathbf{d_{model}}$	**Size in FP32 (GB)**
BERT$_{\text{base}}$	110M	12	768	0.4
BART$_{\text{large}}$ encoder	205M	12	1024	0.8
RoBERTa$_{\text{large}}$	355M	24	1024	1.4
OPT$_{\text{1.3B}}$	1.3B	24	2048	5.2
OPT$_{\text{6.7B}}$	6.7B	32	4096	25.6

```
 1 def backward_pass(self, y1, y2, dy1, dy2):
 2     with torch.enable_grad():
 3         y1.requires_grad = True
 4         # The intermediate activations of G are stored
 5         g_y1 = self.G(y1)
 6         # Obtain the gradient of y1
 7         g_y1.backward(dy2, retain_graph=True)
 8 
 9     with torch.no_grad():
10         x2 = (y2 - g_y1) / self.x2_factor
11         # Save memory, same for below
12         del g_y1, y2
13         dy1 += y1.grad
14         # Save memory
15         y1.grad = None
16 
17     with torch.enable_grad():
18         x2.requires_grad = True
19         # The intermediate activations of F are stored
20         f_x2 = self.F(x2)
21         # Obtain the gradient of x2
22         f_x2.backward(dy1, retain_graph=False)
23 
24     with torch.no_grad():
25         x1 = (y1 - f_x2) / self.x1_factor
26         del f_x2, y1
27         dy2 *= self.x2_factor
28         # dy2=dx2, save memory by using the same variable
29         dy2 += x2.grad
30         x2.grad = None
31         # dy1=dx1
32         dy1 *= self.x1_factor
33         x2 = x2.detach()
34     return x1, x2, dy1, dy2
```

Listing 1: Backward pass for each Layer. The peak memory happens at Line 10 or Line 25, depending on whether the subnetwork $\mathcal{G}$ is larger than $\mathcal{F}$ or the opposite. In the code, we use x1, x2, y1, y2, x1_factor, x2_factor to represent $\boldsymbol{h}_{n-1}^1$, $\boldsymbol{h}_{n-1}^2$, $\boldsymbol{h}_n^1$, $\boldsymbol{h}_n^2$, λ and β, respectively.

Table 10: Compared to $\mathcal{Y}$-Tuning on RoBERTa$_{\text{large}}$. We exclude the memory of $\mathcal{Y}$-Tuning for BART in Table 2, because it was not reported. Instead, the memory usage of $\mathcal{Y}$-Tuning for RoBERTa$_{\text{large}}$ was reported. Notably, the STS-B task is excluded from the calculation of the average score, because it was not evaluated in Liu et al. [20].

Model	**#Parameter**	**Peak Memory (GB)**	**Average Score**
Full FT	100%	11.47	**88.4**
LoRA	**0.23%**	6.11	88.1
$\mathcal{Y}$-Tuning	4.57%	2.08	82.1
MEFT$_1$	**0.23%**	3.63	**88.4**

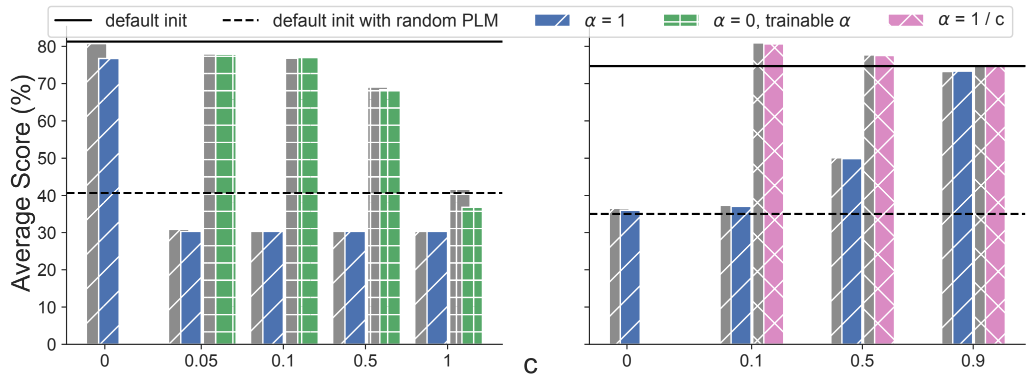

Figure 10: The initialization effect for PEFT, Left: LoRA, Right: (IA)^3. Instead of initializing $\boldsymbol{W}_{up} = \boldsymbol{c}$ like Figure 2b, here we initialize it as $\boldsymbol{W}_{up} \sim \mathcal{N}(c, 0.02^2)$, which should be more suitable for training due to its asymmetry. For convenient comparison, the results of $\boldsymbol{W}_{up} = \boldsymbol{c}$ (in grey) are also included. Overall, the results between $\boldsymbol{W}_{up} = \boldsymbol{c}$ and $\boldsymbol{W}_{up} \sim \mathcal{N}(c, 0.02^2)$ are comparable. However, when $c = 0$ for LoRA, the result of Gaussian initialization is slightly worse than the constant initialization. This further supports our starting point hypothesis, since the Gaussian initialization can't guarantee the output from the adapter is strictly equal to zero at the beginning of fine-tuning.