COMPACTER: Efficient Low-Rank Hypercomplex Adapter Layers

Rabeeh Karimi Mahabadi
EPFL University, Idiap Research Institute
rabeeh.karimi@idiap.ch

James Henderson
Idiap Research Institute
james.henderson@idiap.ch

Sebastian Ruder
DeepMind
ruder@google.com

Abstract

Adapting large-scale pretrained language models to downstream tasks via fine-tuning is the standard method for achieving state-of-the-art performance on NLP benchmarks. However, fine-tuning all weights of models with millions or billions of parameters is sample-inefficient, unstable in low-resource settings, and wasteful as it requires storing a separate copy of the model for each task. Recent work has developed *parameter-efficient* fine-tuning methods, but these approaches either still require a relatively large number of parameters or underperform standard fine-tuning. In this work, we propose COMPACTER, a method for fine-tuning large-scale language models with a better trade-off between task performance and the number of trainable parameters than prior work. COMPACTER accomplishes this by building on top of ideas from adapters, low-rank optimization, and parameterized hypercomplex multiplication layers.

Specifically, COMPACTER inserts task-specific weight matrices into a pretrained model's weights, which are computed efficiently as a sum of Kronecker products between shared "slow" weights and "fast" rank-one matrices defined per COMPACTER layer. By only training 0.047% of a pretrained model's parameters, COMPACTER performs on par with standard fine-tuning on GLUE and outperforms standard fine-tuning on SuperGLUE and low-resource settings. Our code is publicly available at https://github.com/rabeehk/compacter.

1 Introduction

> With four parameters I can fit an elephant,
> and with five I can make him wiggle his trunk.
>
> John von Neumann

State-of-the-art pretrained language models (PLMs) in natural language processing (NLP) have used heavily over-parameterized representations consisting of hundreds of millions or billions of parameters to achieve success on a wide range of NLP benchmarks [2, 3, 4]. These models are generally applied to downstream tasks via fine-tuning [5], which requires updating *all* parameters and storing one copy of the fine-tuned model per task. This causes substantial storage and deployment costs and hinders the applicability of large-scale PLMs to real-world applications. Additionally, fine-tuning of over-parameterized models on low-resource datasets has been shown to be subject to instabilities and may lead to poor performance [6, 7].

Inspired by John von Neumann's quotation, we ask, given that we have already learned general-purpose language representations via a PLM (i.e. we have fit our elephant), how many more parameters

35th Conference on Neural Information Processing Systems (NeurIPS 2021)

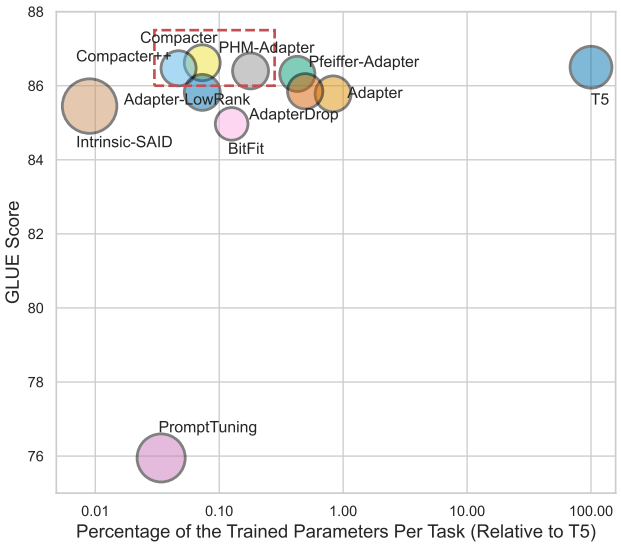

Figure 1: The average score on GLUE (y axis), percentage of trainable parameters per task (x axis, in log scale), and memory footprint (size of the circles) of different methods.

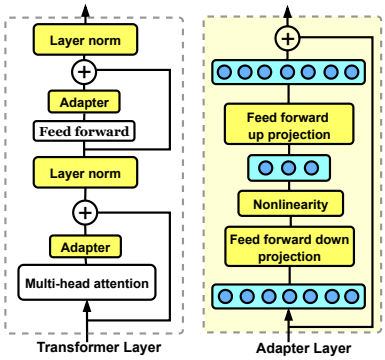

Figure 2: Left: Adapter integration in a pretrained transformer model. Right: Adapter architecture. Following Houlsby et al. [1], we include adapters after the attention and feed-forward modules. During training, we only update layer normalizations and adapters (shown in yellow), while the pretrained model is fixed.

do we need to reach state-of-the-art performance on standard NLP tasks. Specifically, we aim to develop practical, memory-efficient methods that train a minimum set of parameters while achieving performance on par or better than full fine-tuning for state-of-the-art NLP models.

Recent literature has introduced *parameter-efficient* fine-tuning methods. These approaches generally keep the pretrained model's parameters fixed and introduce a set of trainable parameters per task, trading off the number of trainable parameters with task performance. At one end of the spectrum, *prompts*, i.e. natural language descriptions of a task, together with demonstrations have been used to achieve reasonable performance *without any* parameter updates on some benchmarks [8] but their performance generally lags behind fine-tuned models. They also require huge models to work well but choosing good prompts becomes harder with larger model sizes [9]. *Soft prompt* methods treat prompts as trainable continuous parameters, which are prepended to the inputs at the input layer or intermediate layers [10, 11, 12]. Such methods, however, often require large models to achieve good performance and are very sensitive to initialization and unstable during training.

The theoretically motivated *low-rank* methods train a small number of parameters that lie in a low-dimensional subspace using random projections [13, 14]. However, storing the random projection matrices causes substantial memory overhead and leads to slow training times. At the other end of the spectrum, *adapter* methods [1, 15] that insert trainable transformations at different layers of the pretrained model require more parameters than the aforementioned approaches but are more memory-efficient and obtain performance comparable to full fine-tuning [1, 16].

In this work, we propose COMPACTER, a method for fine-tuning large-scale language models with an excellent trade-off between the number of trainable parameters, task performance, and memory footprint, compared to existing methods (see Figure 1). COMPACTER builds on ideas from adapters [1], low-rank methods [13], as well as recent hypercomplex multiplication layers [17]. Similar to adapters, COMPACTER inserts task-specific weight matrices into a pretrained model's weights. Each COMPACTER weight matrix is computed as the sum of Kronecker products between shared "slow" weights and "fast" rank-one matrices defined per COMPACTER layer (see Figure 3). As a result, COMPACTER achieves a parameter complexity of $\mathcal{O}(k+d)$ compared to $\mathcal{O}(kd)$ for regular adapters, where the adapters are of size $k \times d$. In practice, COMPACTER trains 0.047% of a PLM's parameters. On the standard GLUE [18] and SuperGLUE [19] benchmarks, COMPACTER outperforms other parameter-efficient fine-tuning methods and obtains performance on par or better than full fine-tuning. On low-resource settings, COMPACTER outperforms standard fine-tuning.

In summary, we make the following contributions: **1)** We propose COMPACTER (**Compact** Adapt**er**) layers, a parameter-efficient method to adapt large-scale language models. **2)** We show that COMPACTER obtains strong empirical performance on GLUE and SuperGLUE. **3)** We demonstrate that

COMPACTER outperforms fine-tuning in low-resource settings. **4)** We provide a parameter complexity analysis of COMPACTER, showing that it requires dramatically fewer parameters than adapters and fine-tuning. **5)** We provide a systematic evaluation of recent parameter-efficient fine-tuning methods in terms of training time and memory consumption. We release our code to facilitate future work.

2 Background

We start by introducing the required background on the Kronecker product and adapter layers [1, 15].

2.1 Kronecker Product

The Kronecker product between matrix $\boldsymbol{A} \in \mathbb{R}^{m \times f}$ and $\boldsymbol{B} \in \mathbb{R}^{p \times q}$, denoted by $\boldsymbol{A} \otimes \boldsymbol{B} \in \mathbb{R}^{mp \times fq}$, is mathematically defined as:

$$\boldsymbol{A} \otimes \boldsymbol{B} = \begin{pmatrix} a_{11}\boldsymbol{B} & \cdots & a_{1f}\boldsymbol{B} \\ \vdots & \ddots & \vdots \\ a_{m1}\boldsymbol{B} & \cdots & a_{mf}\boldsymbol{B} \end{pmatrix}, \tag{1}$$

where a_{ij} shows the element in the ith row and j^{th} column of $\boldsymbol{A}$.

2.2 Adapter Layers

Recent work has shown that fine-tuning *all* parameters of a language model can lead to a sub-optimal solution, particularly for low-resource datasets [6]. As an alternative, Rebuffi et al. [15] and Houlsby et al. [1] propose to transfer a model to new tasks by inserting small task-specific modules called *adapter layers* within the layers of a pretrained model, as depicted in Figure 2. They then only train adapters and layer normalizations, while the remaining parameters of the pretrained model remain fixed. This approach allows pretrained language models to efficiently adapt to new tasks.

Each layer of a transformer model is composed of two primary modules: a) an attention block, and b) a feed-forward block. Both modules are followed by a skip connection. As shown in Figure 2, Houlsby et al. [1] suggest to insert an adapter layer after each of these blocks before the skip connection.

Adapters are bottleneck architectures. By keeping the output dimension similar to their input, they cause no change to the structure or parameters of the original model. The adapter layer A^l for layer l consists of a down-projection, $\boldsymbol{D^l} \in \mathbb{R}^{k \times d}$, GeLU non-linearity [20], and up-projection $\boldsymbol{U^l} \in \mathbb{R}^{d \times k}$, where k is the input dimension, and d is the bottleneck dimension for the adapter layer. Adapters are defined as:

$$A^l(\boldsymbol{x}) = \boldsymbol{U^l}(\text{GeLU}(\boldsymbol{D^l}(\boldsymbol{x}))) + \boldsymbol{x}, \tag{2}$$

where $\boldsymbol{x}$ is the input hidden state.

3 Method

In this section, we present COMPACTER, a compact and efficient way to adapt large-scale PLMs.

Problem formulation We consider the general problem of fine-tuning large-scale language models, where we are given the training data $\mathcal{D} = \{(\boldsymbol{x^i}, y^i)\}_{i=1}^{P}$ with P samples. We assume we are also given a large-scale pretrained language model $f_{\boldsymbol{\theta}}(.)$ parameterized by $\boldsymbol{\theta}$ that computes the output for input $\boldsymbol{x^i}$. Our goal is to fine-tune $f_{\boldsymbol{\theta}}(.)$ efficiently to enable the model to adapt to new tasks.

3.1 Compact and Efficient Adapter Layers

In this section, we introduce an efficient version of adapter layers, building on top of recent advances in *parameterized hypercomplex multiplication layers* (PHM) [17]. To the best of our knowledge, we are the first to exploit PHM layers for efficient fine-tuning of large-scale transformer models. The PHM layer has a similar form as a fully-connected layer, which converts an input $\boldsymbol{x} \in \mathbb{R}^k$ to an output $\boldsymbol{y} \in \mathbb{R}^d$:

$$\boldsymbol{y} = \boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}, \tag{3}$$

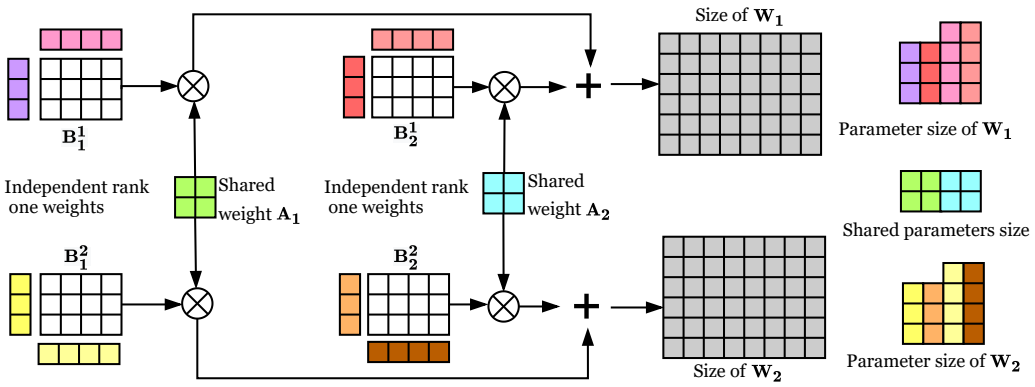

Figure 3: Illustration of generating weights of two different COMPACTER layers: $\boldsymbol{W_1} \in \mathbb{R}^{d\times k}$ (first row) and $\boldsymbol{W_2} \in \mathbb{R}^{d\times k}$ (second row). We generate $\boldsymbol{W_1}$ and $\boldsymbol{W_2}$ using $\boldsymbol{W_j} = \sum_{i=1}^{n} \boldsymbol{A_i} \otimes \boldsymbol{B_i}^{\boldsymbol{j}} = \sum_{i=1}^{n} \boldsymbol{A_i} \otimes (\boldsymbol{s_i}^{\boldsymbol{j}} \boldsymbol{t_i}^{\boldsymbol{j}\top})$ (5), by computing the sum of Kronecker products of *shared* matrices $\boldsymbol{A_i}$ and *adapter-specific* matrices $\boldsymbol{B_i^j}$, with $i \in \{1,...,n\}$ and adapter index $j \in \{1,2\}$. We generate each $\boldsymbol{B_i^j}$ by multiplying independent rank one weights. In this example $n=2$, $d=6$, and $k=8$.

where $\boldsymbol{W} \in \mathbb{R}^{k\times d}$. The key difference is that in a PHM layer, $\boldsymbol{W}$ is learned as a sum of Kronecker products. Assume that k and d are both divisible by a user-defined hyperparameter $n \in \mathbb{Z}_{>0}$. Then, the matrix $\boldsymbol{W}$ in (3) is computed as the sum of n Kronecker products as follows:

$$\boldsymbol{W} = \sum_{i=1}^{n} \boldsymbol{A_i} \otimes \boldsymbol{B_i}, \tag{4}$$

where $\boldsymbol{A_i} \in \mathbb{R}^{n\times n}$ and $\boldsymbol{B_i} \in \mathbb{R}^{\frac{k}{n}\times\frac{d}{n}}$. The PHM layer has a parameter complexity of $\mathcal{O}(\frac{kd}{n})$, reducing parameters by at most $\frac{1}{n}$ [17] (see §4).

3.2 Beyond Hypercomplex Adapters

Prior work indicates that some of the information captured in pretrained models can be ignored for transfer [21, 22]. Similarly, redundancies have been observed in the information captured by adapters, with adapters in lower layers being less important [1]. In addition, sharing adapters across layers leads to a comparatively small drop of performance for some tasks [23]. Motivated by these insights, we propose the following two extensions to make hypercomplex adapters more efficient.

Sharing information across adapters Sharing all adapter parameters across layers is overall too restrictive and is not able to perform on par with fine-tuning or using regular adapters [23]; however, our decomposition of adapters into $\boldsymbol{A_i}$ and $\boldsymbol{B_i}$ matrices as in Eq. (4) allows us to be more flexible. Consequently, we divide our adaptation weights into *shared* parameters that capture general information useful for adapting to the target task and *adapter-specific* parameters that focus on capturing information relevant for adapting each individual layer. Specifically, we define $\boldsymbol{A_i}$ as shared parameters that are common across all adapter layers while $\boldsymbol{B_i}$ are adapter-specific parameters.

Low-rank parameterization Low-rank methods [13, 14] have demonstrated that strong performance can be achieved by optimizing a task in a low-rank subspace. Similarly, we hypothesize that a model can also be effectively adapted by learning transformations in a low-rank subspace. To this end, we propose to parameterize $\boldsymbol{B_i} \in \mathbb{R}^{\frac{k}{n}\times\frac{d}{n}}$ as a low-rank matrix, which is the product of two low-rank weights $\boldsymbol{s_i} \in \mathbb{R}^{\frac{k}{n}\times r}$ and $\boldsymbol{t_i} \in \mathbb{R}^{r\times\frac{d}{n}}$, where r is the rank of the matrix.[1] Putting both extensions together, we propose the *low-rank* parameterized hypercomplex multiplication layer (LPHM):

$$\boldsymbol{W} = \sum_{i=1}^{n} \boldsymbol{A_i} \otimes \boldsymbol{B_i} = \sum_{i=1}^{n} \boldsymbol{A_i} \otimes (\boldsymbol{s_i}\boldsymbol{t_i}^{\top}). \tag{5}$$

In general, we set $r=1$ so that $\boldsymbol{B_i}$ is a rank-one matrix. Depending on the complexity of the target task, r can be set to a higher value.[2] Figure 3 illustrates our method. Overall, the LPHM layer reduces

[1] We do not factorize $\boldsymbol{A_i}$ as they are small, shared between all layers, and factorization hurts performance.

[2] If factors are over-parameterized, COMPACTER can be used for *overcomplete* knowledge distillation [24].

complexity further to $\mathcal{O}(k+d)$ (see §4). The LPHM layer can also be seen as leveraging "slow" weights $\boldsymbol{A_i}$ that are shared across adapters and capture general information and "fast" weights $\boldsymbol{B_i}$ that learn adapter-specific information for adaptation of each individual layer [25].

COMPACTER Based on the above formulation, we introduce COMPACTER layers, which replace the down-projection and up-projection layers in adapters as follows:

$$A^l(\boldsymbol{x}) = \text{LPHM}^{U^l}(\text{GeLU}(\text{LPHM}^{D^l}(\boldsymbol{x}))) + \boldsymbol{x},$$

where the up-projection weights LPHM^{U^l} are computed as in (5), replacing the layer U^l in (2). Similarly, down-projection weights LPHM^{D^l} replace the layer D^l. While the two adapters in each layer of a transformer have their own $\boldsymbol{s_i}$ and $\boldsymbol{t_i}$ rank-one weights, we share the $\boldsymbol{A_i}$ across all layers and positions of the adapter layers.

4 Parameter Efficiency

In this section, we compare the number of parameters of COMPACTER with adapters.

Adapters parameters In the standard setting, two adapters are added per layer of a transformer model [1]. Each adapter layer consists of $2kd$ parameters for the down and up-projection matrices ($\boldsymbol{U^l}$, $\boldsymbol{D^l}$) respectively where k is the size of the input dimension and d is the adapter's bottleneck dimension. The total number of parameters for adapters for a transformer model with L layers of both an encoder and a decoder is, therefore, $2L(2kd)$, which scales linearly with all three variables.

PHM-ADAPTER parameters In the conventional PHM layer [17], as depicted in Eq. (4), parameters of $\boldsymbol{A_i} \in \mathbb{R}^{n \times n}$ and $\boldsymbol{B_i} \in \mathbb{R}^{\frac{k}{n} \times \frac{d}{n}}$ define the degree of freedom for $\boldsymbol{W}$ as $n(\frac{kd}{n^2} + n^2) = \frac{kd}{n} + n^3$. With the mild condition that $kd > n^4$, then $\frac{kd}{n}$ dominates and the overall parameter size of the PHM layer in (4) is $\mathcal{O}(\frac{kd}{n})$. This condition is satisfied for typical values for adapters, PHM layers, and large-scale PLMs such as T5-large, with hidden size $k = 1024$, adapter hidden size $d \in \{24,32,48,96\}$, and $n = 2,4,8,12$. Hence, the PHM layer offers a parameter reduction of almost $\frac{1}{n}$ compared to standard fully-connected layers, which are $\mathcal{O}(kd)$.[3]

Similarly, employing PHM layers for modeling down and up-projection matrices offers a parameter reduction of almost $\frac{1}{n}$. Each adapter with a PHM layer has in total $2(\frac{kd}{n} + n^3)$ parameters. For a Transformer model with L layers, the total number of parameters of PHM-ADAPTER is $4L(\frac{kd}{n} + n^3)$.

COMPACTER parameters COMPACTER shares the trained weight matrices $\{\boldsymbol{A_i}\}_{i=1}^{n}$ in (5) consisting of n^3 parameters across all layers. COMPACTER also has two rank-one weights for each adapter, $\boldsymbol{s_i}, \boldsymbol{t_i}$ in (5) consisting of $\frac{k}{n} + \frac{d}{n}$ parameters, resulting in a total of $2n(\frac{k}{n} + \frac{d}{n})$ parameters for down and up-projection weights. Therefore, the total number of parameters of COMPACTER is $4L(k+d) + n^3$ for a transformer with L layers in the encoder and decoder.

In settings with a large number of layers, the dominant term is $4L(k+d)$. Therefore, with a mild condition that $4L(k+d) > n^3$, COMPACTER has a complexity of $\mathcal{O}(k+d)$, which is far more efficient compared to adapters' $\mathcal{O}(kd)$ and PHM-ADAPTER's $\mathcal{O}(\frac{kd}{n})$ complexity respectively. In settings where n is large, the number of parameters for shared weight matrices $\{\boldsymbol{A_i}\}_{i=1}^{n}$ for all layers remain constant in COMPACTER with a total of n^3 parameters while this scales linearly with the number of layers L for PHM and adapter layers. As an example, in the T5_{BASE} model with 222M parameters [3], COMPACTER only learns 0.047% of the parameters, and maintains comparable performance to *full fine-tuning*.

5 Experiments

Datasets Following Raffel et al. [3], we evaluate the performance of the methods on the GLUE [18] and SUPERGLUE [19] benchmarks. These benchmarks cover multiple tasks of paraphrase detection (MRPC, QQP), sentiment classification (SST-2), natural language inference (MNLI, RTE, QNLI, CB), linguistic acceptability (CoLA), question-answering (MultiRC, ReCoRD, BoolQ), word sense disambiguation (WiC), and sentence completion (COPA).[4] As the original test sets are not publicly

[3] Even for smaller models where the n^4 term dominates, we observe a substantial reduction of parameters compared to adapters.

[4] Following Devlin et al. [2], Raffel et al. [3], as a common practice, we do not experiment with WNLI [26] due to its adversarial nature with respect to the training set.

available, we follow Zhang et al. [27] and split off 1k samples from the training set that we use for validation, while we use the original validation data as the test set. For datasets with fewer than 10k samples (RTE, MRPC, STS-B, CoLA, COPA, WiC, CB, BoolQ, MultiRC), we divide the original validation set in half, using one half for validation and the other for testing.

Experimental details We use the state-of-the-art encoder-decoder T5 model [3] as the underlying model for all methods in our experiments. For computational efficiency, we report all results on T5_{BASE} models (12 encoder and decoder layers and 222M parameters). We use its HuggingFace PyTorch implementation [28]. We fine-tune all methods for 3 epochs on large datasets and 20 epochs for low-resource datasets of GLUE (MRPC, CoLA, STS-B, RTE, BoolQ, CB, COPA, WiC) to allow the models to converge [27]. For all adapter-based methods, we experiment with adapters of bottleneck size of $\{96,48,24\}$. We save a checkpoint every epoch for all models and report the results for the hyper-parameters performing the best on the validation set for each task. For the PHM layers, we use the PyTorch implementation of Le et al. [29]. We include low-level details in Appendix A. For our methods, we experiment with $n=\{4,8,12\}$ and report the model performing the best. We include the results for all values of n in Appendix B.

Following Mahabadi et al. [30], we freeze the output layer of the pretrained model for all tasks across all methods.[5] We show the results with fine-tuning the output layer in Appendix C. Following Houlsby et al. [1], we update the layer normalization parameters for all methods where applicable.[6]

5.1 Baselines

We compare against several recently proposed *parameter-efficient* fine-tuning methods:

T5_{BASE} We compare our method to the standard practice of fine-tuning T5, where we fine-tune all parameters of the model on each individual task.

ADAPTER We compare to a strong adapter baseline [1], which adds adapters for each task after the feed-forward and attention modules in each transformer block of T5.

PFEIFFER-ADAPTER Pfeiffer et al. [31] propose a more efficient adapter variant, which keeps only one of the adapters in each layer for better training efficiency. We experimented with keeping either adapter and found keeping the adapter after the self-attention module in each layer to perform the best.

ADAPTER-LOWRANK We parameterize each adapter's weight as a product of two rank-one weights.

PROMPT TUNING Prompt tuning [12] is the successor variant of Li and Liang [10], which prepends a randomly initialized continuous prompt to the input (PROMPT TUNING-R). We also compare to a variant, which initializes prompts using token embeddings of the pretrained language model's vocabulary (PROMPT TUNING-T) [12].

INTRINSIC-SAID The Structure Aware Intrinsic Dimension [14] fine-tunes the model by reparameterizing the parameters in a lower-dimensional subspace $\boldsymbol{\theta}^{\boldsymbol{d'}}$ ($d' \ll D$): $\boldsymbol{\theta}_{\boldsymbol{i}}^{\boldsymbol{D}} = \boldsymbol{\theta}_{\boldsymbol{i,0}}^{\boldsymbol{D}} + \lambda_i \boldsymbol{P} \boldsymbol{\theta}_{\boldsymbol{i}}^{\boldsymbol{d'-m}}$ where parameter $\boldsymbol{\theta}_{\boldsymbol{i,0}}^{\boldsymbol{D}}$ are the pretrained model's parameters and $\boldsymbol{P} \in \mathbb{R}^{d'-m} \rightarrow \mathbb{R}^{D}$ is a random linear projection via the Fastfood transform [32]. They then consider the total number of weight matrices in the PLM, m, and attribute a weight to each of them, resulting in $\boldsymbol{\lambda} \in \mathbb{R}^{m}$ in total by trading m parameters from the low dimensional space $\boldsymbol{\theta}^{\boldsymbol{d'}} \in \mathbb{R}^{d'}$. Then, the total trainable parameters are $\boldsymbol{\theta}^{\boldsymbol{d'-m}} \in \mathbb{R}^{d'-m}$ and $\boldsymbol{\lambda}$.

ADAPTERDROP We apply the method of Rücklé et al. [23], which drops the adapters from lower transformer layers for a better training efficiency to T5 with ADAPTER. Consequently, we drop adapters from the first five layers of both the encoder and the decoder in T5_{BASE}.

BITFIT Cai et al. [33] propose to freeze the weights and only train the biases. By not storing intermediate activations, this method enables substantial memory savings. Ravfogel et al. [34] study a similar method for PLMs that fine-tunes only the biases and the final output layer.[7]

[5] This is much more efficient as the output layer includes 11.1% of the parameters of T5_{BASE}. Tasks are formulated in a text-to-text format so the model can be applied to them without learning a new output layer [3]. We note that this is in contrast to the original adapter setting, which used an encoder-only masked PLM [1].

[6] For BITFIT, we only update the biases. For PROMPT TUNING, the entire model is frozen.

[7] Note that in the HuggingFace T5 implementation, the biases in layer normalizations, linear layers, the output layer and self-attention layers are removed. We re-introduce these biases for BITFIT.

Table 1: Performance of all models on the GLUE tasks. For each method, we report the total number of parameters across all tasks and the number of parameters that are trained for each task as a multiple and proportion of $T5_{BASE}$ model [3]. For MNLI, we report accuracy on the matched validation set. For MRPC and QQP, we report accuracy and F1. For STS-B, we report Pearson and Spearman correlation coefficients. For CoLA, we report Matthews correlation. For all other tasks, we report accuracy. Bold fonts indicate the best results. For the results with †, due to insatiability during training, we restarted experiments with 6 random seeds and report the best. For INTRINSIC-SAID, d' is set to 20K.

Method	#Total params	Trained params / per task	CoLA	SST-2	MRPC	QQP	STS-B	MNLI	QNLI	RTE	Avg
Baselines											
$T5_{BASE}$	8.0×1	100%	61.76	**94.61**	**90.20/93.06**	**91.63/88.84**	89.68/89.97	**86.78**	93.01	71.94	**86.50**
ADAPTER	1.065	0.832%	**64.02**	93.81	85.29/89.73	90.18/87.20	90.73/91.02	86.49	93.21	71.94	85.78
PFEIFFER-ADAPTER	1.032	0.427%	62.9	93.46	86.76/90.85	90.14/87.15	91.13/91.34	86.26	93.30	**76.26**	86.32
ADAPTERDROP	1.038	0.494%	62.7	93.58	86.27/90.60	90.2/87.25	**91.37/91.61**	86.27	93.23	71.22	85.85
ADAPTER-LOWRANK	1.004	0.073%	59.19	93.69	88.24/91.49	90.23/87.01	90.8/91.33	85.8	92.9	73.38	85.82
PROMPT TUNING-R	1.003	0.034%	0.47†	87.61	68.14/81.05	88.93/85.55	90.25/90.59	46.83†	92.33	54.68	71.49
PROMPT TUNING-T	1.003	0.034%	10.59	90.94	68.14/81.05	89.69/86.14	89.84/90.21	81.46	92.75	54.68	75.95
INTRINSIC-SAID	1.001	0.009%	58.69	94.15	88.24/91.78	90.28/87.13	90.06/90.45	85.23	**93.39**	70.50	85.45
BITFIT	1.010	0.126%	58.16	94.15	86.76/90.53	90.06/86.99	90.88/91.26	85.31	92.99	67.63	84.97
Our Proposed Methods											
PHM-ADAPTER ($n=12$)	1.013	0.179%	57.35	**94.50**	**91.67/93.86**	**90.25/87.05**	90.45/90.84	**85.97**	92.92	75.54	86.40
COMPACTER ($n=4$)	1.004	0.073%	**63.75**	93.00	89.22/92.31	90.23/87.03	90.31/90.74	85.61	92.88	**77.70**	**86.62**
COMPACTER++ ($n=4$)	1.002	0.047%	61.27	93.81	90.69/93.33	90.17/86.93	**90.46/90.93**	85.71	**93.08**	74.82	86.47

5.2 Our Methods

PHM-ADAPTER We learn the weights of adapters using PHM layers as in (4). To our knowledge, we are the first who exploit the idea of PHM [17] for efficient *fine-tuning* of large-scale language models.

COMPACTER We learn adapter weights using LPHM layers as described in (5). We also explore a variant where we only keep the COMPACTER layer after the feed-forward layer in each transformer block (COMPACTER++).[8]

5.3 Results on the GLUE Benchmark

Table 1 shows the results on GLUE with $T5_{BASE}$ (see Appendix E for results on $T5_{SMALL}$). COMPACTER and COMPACTER++ outperform all previous parameter-efficient methods and perform on par with full fine-tuning while only training 0.07% and 0.047% of parameters respectively. We now discuss the different methods in detail.

Adapter-based methods For ADAPTER, not fine-tuning the classifier hurts the performance substantially (85.78 versus 86.48; cf. Appendix C). PFEIFFER-ADAPTER, which adds adapters only after the self-attention module outperforms the standard ADAPTER while being more parameter-efficient. ADAPTERDROP obtains lower performance than fine-tuning, demonstrating that adapting the lower layers of an encoder-decoder T5 model is important for its performance. Additionally, ADAPTER-LOWRANK is not expressive enough to perform well on this benchmark.

Prompt tuning and BitFit For PROMPT TUNING, we observe high sensitivity to initialization and learning rate, as also confirmed in [10]. We experimented with multiple random seeds but performance lags behind fine-tuning substantially, in particular on low-resource datasets. This can be explained by the low flexibility of such methods as all the information needs to be contained in the prefixes. As a result, the method only allows limited interaction with the rest of the model and good performance requires very large models [12]. In addition, increasing the sequence length leads to memory overhead (see §5.5) and the number of prompt tokens is limited by the number of tokens that can fit in the model's maximum input length, which makes such methods less flexible and unsuitable for dealing with large contexts. Similarly, BITFIT performs worse than fine-tuning, especially on low-resource datasets.

Intrinsic-SAID Interestingly, the average performance of INTRINSIC-SAID, which fine-tunes only 0.009% of a model's parameters is only 1.05 points below the fine-tuning baseline. However, this method has two practical drawbacks: a) storing the random projection matrices results in a substantial

[8] We found this to slightly outperform keeping the COMPACTER layer after the self-attention layer instead.

Table 2: Performance of all methods on the SUPERGLUE tasks. For each method, we report the total number of parameters across all tasks and the percentage of parameters that are trained for each task as a multiple and proportion of $T5_{BASE}$ model [3]. For CB, we report accuracy and F1. For MultiRC, we report F1 over all answer-options ($F1_a$) and exact match of each question's set of answers (EM) [19]. For ReCoRD, we report F1 and EM scores. For all other tasks, we report accuracy. For INTRINSIC-SAID, d' is set to 20K. Bold fonts indicate the best results in each block.

Method	#Total params	Trained params / per task	BoolQ	CB	COPA	MultiRC	ReCoRD	WiC	Avg
Baselines									
$T5_{BASE}$	6.0×1	100%	81.10	**85.71/78.21**	52.0	68.71/47.0	74.26/73.33	**70.22**	70.06
ADAPTER	1.049	0.832%	82.39	85.71/73.52	52.0	72.75/53.41	74.55/73.58	67.08	70.55
PFEIFFER-ADAPTER	1.024	0.427%	**82.45**	85.71/75.63	54.0	72.53/51.76	74.69/73.70	68.65	**71.01**
ADAPTERDROP	1.028	0.494%	82.26	85.71/75.63	42.0	**72.92/53.30**	74.68/73.70	68.34	69.84
ADAPTER-LOWRANK	1.003	0.073%	80.31	78.57/55.37	54.0	72.58/51.98	74.77/73.87	64.58	67.34
PROMPT TUNING-R	1.002	0.034%	61.71	67.86/46.99	48.0	59.23/16.33	75.27/74.36	48.90	55.41
PROMPT TUNING-T	1.002	0.034%	61.71	67.86/46.89	52.0	57.66/19.44	**75.37/74.41**	48.90	56.03
INTRINSIC-SAID	1.001	0.009%	78.72	75.00/51.83	54.0	69.98/52.78	74.86/73.91	65.83	66.32
BITFIT	1.008	0.126%	79.57	78.57/54.40	**56.0**	70.73/48.57	74.64/73.64	69.59	67.30
Our Proposed Methods									
PHM-ADAPTER ($n=4$)	1.013	0.240%	**80.31**	85.71/73.52	44.0	**71.99/51.65**	**74.62/73.60**	67.40	69.20
COMPACTER ($n=12$)	1.003	0.073%	78.59	**96.43/87.44**	48.0	70.80/49.67	74.49/73.54	65.20	71.57
COMPACTER++ ($n=12$)	1.002	0.048%	78.84	92.86/84.96	**52.0**	70.68/50.99	74.55/73.50	**68.03**	**71.82**

memory overhead; b) it is very slow to train (see §5.5). Despite this, INTRINSIC-SAID provides insights regarding the effectiveness of low-rank optimization of pretrained language models [14], which motivates the development of parameter-efficient methods such as COMPACTER.

COMPACTER For our proposed methods, we observe fine-tuning the output layer for both PHM-ADAPTER and COMPACTER++ does not provide much performance difference (see Appendix C). PHM-ADAPTER reduces the parameters of ADAPTER from 0.83% to 0.179% (with $n=12$), being $4.64\times$ more parameter-efficient. COMPACTER reduces the number of parameters to the remarkable rate of 0.073% while obtaining comparable results to full fine-tuning. By removing the COMPACTER layer after self-attention, COMPACTER++ obtains similar performance, while reducing the parameters to 0.047%. Adaptation without updating the layer normalization can be a promising direction to reduce the parameters further, for instance by building on recent advances in normalization-free models [35], which we leave to future work.

5.4 Results on the SUPERGLUE Benchmark

Table 2 shows the performance of the methods on SUPERGLUE [19]. We include the results for all values of n in Appendix D. We observe a similar pattern as on GLUE in Table 1. COMPACTER and COMPACTER++ perform substantially better compared to other parameter-efficient fine-tuning methods and even outperform full fine-tuning while only training 0.073% and 0.048% of the parameters.

5.5 Efficiency Evaluation

In this section, we compare the efficiency of our proposed methods with various recently proposed parameter-compact fine-tuning methods under the same computation budget. To this end, we train all methods for 1 epoch on the MNLI dataset. For each method, we select the largest batch size that fits a fixed budget of the GPU memory (24 GB). For all adapter-based methods, we fix the adapter size to 24. For PROMPT TUNING, we set the number of prefix tokens to 100. For INTRINSIC-SAID, we set $d' = 1400$. Finally, we set $n = 4$. In Table 3, we report the percentage of trained parameters per task, training time per epoch, and memory usage of each method. Moreover, Figure 1 shows the trade-off between quantitative performance, percentage of trained parameters, and memory footprint.

Our approaches have several attractive properties. Based on our analysis in Table 1, COMPACTER and COMPACTER++ obtain the best combination of high GLUE score averaged across all tasks, plus a substantially lower number of parameters (0.073% and 0.047% respectively). In addition to COMPACTER++

Table 3: Percentage of trained parameters per task, average peak memory and training time for all methods. $\Delta\%$ is the relative difference with respect to *full fine-tuning* ($T5_{BASE}$). Lower is better.

Method	**Trained params/ per task**	**Memory (MB)**	$\Delta\%$	**Time/Epoch (min)**	$\Delta\%$
$T5_{BASE}$	100%	167.99	—	42.13	—
ADAPTER	0.832%	124.02	-35.45%	31.81	-24.50%
PFEIFFER-ADAPTER	0.427%	118.4	-41.88%	28.19	-33.09%
ADAPTERDROP	0.494%	119.41	-40.68%	28.08	-33.35%
ADAPTER-LOWRANK	0.073%	123.8	-35.69%	32.71	-22.36%
PROMPT TUNING	0.034%	222.27	24.42%	44.54	5.72%
INTRINSIC-SAID	0.009%	285.40	41.14%	144.01	241.82%
BITFIT	0.126%	102.31	-64.20%	27.36	-35.06%
PHM-ADAPTER	0.179%	123.93	-35.55%	35.55	-15.62%
COMPACTER	0.073%	123.91	-35.57%	36.48	-13.41%
COMPACTER++	0.047%	118.35	-41.94%	30.96	-26.51%

performing well, its memory requirement is the second best among all methods, reducing memory usage by -41.94% compared to $T5_{BASE}$. COMPACTER and COMPACTER++ also speed up training substantially, by -13.41% and -26.51% relative to $T5_{BASE}$. On the other hand, BITFIT, by not storing intermediate activations, has the lowest memory requirement (-64.2% relative to $T5_{BASE}$) and is the fastest (-35.06% relative to $T5_{BASE}$) at the cost of lower quantitative performance (1.53 points lower; see Table 1).

Methods relying on pruning adapters, i.e., PFEIFFER-ADAPTER and ADAPTERDROP reduce the memory overhead and improve training time. However, their number of parameters is almost an order of magnitude more compared to COMPACTER++, with $9.1\times$ and $10.5\times$ more parameters respectively. Moreover, although, PFEIFFER-ADAPTER performs on par with full fine-tuning with a slight degradation (Table 1), ADAPTERDROP obtains a lower performance (-0.65 less on average across all tasks.). We note that dropping adapters from transformer layers is a general technique and could be applied to COMPACTER for improving efficiency even further, which we leave to future work. Similarly, although ADAPTER-LOWRANK reduces the memory overhead and improves the training time, it obtains a lower performance (Table 1) (-0.68 less on average across all tasks.).

At the other end of the spectrum, INTRINSIC-SAID and PROMPT TUNING methods have the lowest number of parameters. However, they both come with high memory overhead (41.14% and 24.42% relative to full fine-tuning ($T5_{BASE}$) respectively), are slowest to train, and their performance substantially lags behind full fine-tuning (see Table 1). For PROMPT TUNING, high memory costs are due to the fact that the computational complexity of self-attention, which requires storing the full attention matrix for gradient computation, scales quadratically with the sequence length [36]. For INTRINSIC-SAID, the high memory requirement is due to storing large random projection matrices, which limits the application of INTRINSIC-SAID for fine-tuning large-scale PLMs. Moreover, computing projections via FastFood transform, although theoretically possible in $O(D\log d')$ [32], is slow in practice even with a CUDA implementation. For pretrained language models with a large number of parameters, allocating random projections for the full parameter space is intractable. While using Fastfood transform partially ameliorates this issue by reducing the memory usage from $\mathcal{O}(Dd')$ to $\mathcal{O}(D)$, the memory issue with such methods remains unresolved.

Overall, given the size of large-scale transformer models with millions and billions of parameters, such as T5 [3], efficient memory usage is of paramount importance for practical applications. COMPACTER and COMPACTER++ offer a great trade-off in terms of performance, memory usage, and training time. With regard to our inspiration of von Neumann's quotation, we thus find that only a comparatively small number of additional parameters are necessary for the practical and efficient adaptation of PLMs.

5.6 Low-resource Fine-tuning

COMPACTER++ has substantially fewer parameters compared to $T5_{BASE}$. In this section, we investigate whether this could help COMPACTER++ to generalize better in resource-limited settings. We subsample each dataset of GLUE for varying sizes in the range $\{100,500,1000,2000,4000\}$. Figure 4 shows the

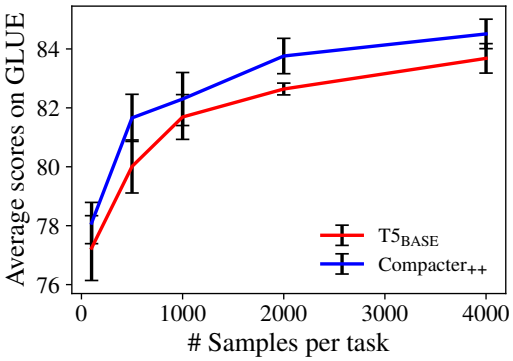

Figure 4: Results on GLUE for the various number of training samples per task $(100,500,1000,2000,4000)$. We show mean and standard deviation across 5 seeds.

results. COMPACTER++ substantially improves the results in the low-resource setting, indicating more effective fine-tuning in this regime.

6 Related Work

Adapters Adapters have recently emerged as a new paradigm for fine-tuning pretrained language models [1]. In another line of work, Üstün et al. [37] proposed a multilingual dependency parsing method based on adapters and contextual parameter generator networks [38], where they generate adapter parameters conditioned on trained input language embeddings. This, however, leads to a large number of additional parameters compared to the base model. Contemporaneously, Mahabadi et al. [30] use a single compact hypernetwork allowing to generate adapter weights efficiently conditioned on multiple tasks and layers of a transformer model. Pilault et al. [39] also proposed a task-conditioned transformer for multi-task learning which is less parameter-efficient. The aforementioned work is complementary to COMPACTER, and one could potentially combine COMPACTER with contextual parameter generation to generate adapter modules. Compared to Mahabadi et al. [30], COMPACTER++ reduces the parameters by 6.2×.

Hypercomplex representations Deep learning advances in the hypercomplex domain are in a nascent stage, and most work is fairly recent [40, 41, 42, 43, 44]. Replacing matrix multiplications in standard networks with Hamilton products that have fewer degrees of freedom offers up to a 4× saving of parameter size in a single multiplication operation [42, 44]. Very recently, Zhang et al. [17] extend such methods in a way that they could reduce the parameters of a fully connected layer under a mild condition to $1/n$, where n is a user-specified parameter. To the best of our knowledge, there is no previous work that attempts to leverage the hypercomplex space for efficient fine-tuning of large-scale language models.

Other parameter-efficient models Li et al. [13] and Aghajanyan et al. [14] study training models in a low-dimensional randomly oriented subspace instead of their original parameter space. Another recent line of work has shown that pretrained models such as BERT are redundant in their capacity, allowing for significant sparsification without much degradation in end metrics [45, 46, 47]. Such methods, however, remain not well supported by current hardware and often perform worse compared to dedicated efficient architectures [48].

7 Conclusion

We have proposed COMPACTER, a light-weight fine-tuning method for large-scale language models. COMPACTER generates weights by summing Kronecker products between shared "slow" weights and "fast" rank-one matrices, specific to each COMPACTER layer. Leveraging this formulation, COMPACTER reduces the number of parameters in adapters substantially from $\mathcal{O}(kd)$ to $\mathcal{O}(k+d)$. Through extensive experiments, we demonstrate that despite learning 2127.66× fewer parameters than standard fine-tuning, COMPACTER obtains comparable or better performance in a full-data setting and outperforms fine-tuning in data-limited scenarios.

Acknowledgements

We are grateful to Dani Yogatama for feedback on a draft of this manuscript. The authors would like to thank Tuan Le for his assistance in reproducing the results of Zhang et al. [17]. We would like to also thank Armen Aghajanyan for his assistance to reproduce the results of his work [14]. We thank Jue Wang for his comments on an earlier version of this paper. The authors are grateful to Brian Lester, Rami Al-Rfou, Noah Constant, and Mostafa Dehghani for their assistance. Rabeeh Karimi Mahabadi was supported by the Swiss National Science Foundation under the project Learning Representations of Abstraction for Opinion Summarization (LAOS), grant number "FNS-30216".

References

[1] 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 *ICML*, 2019.

[2] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. In *NAACL*, 2019.

[3] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *JMLR*, 2020.

[4] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692*, 2019.

[5] Jeremy Howard and Sebastian Ruder. Universal Language Model Fine-tuning for Text Classification. In *ACL*, 2018.

[6] Matthew E Peters, Sebastian Ruder, and Noah A Smith. To tune or not to tune? adapting pretrained representations to diverse tasks. In *RepL4NLP*, 2019.

[7] Jesse Dodge, Gabriel Ilharco, Roy Schwartz, Ali Farhadi, Hannaneh Hajishirzi, and Noah Smith. Fine-Tuning Pretrained Language Models: Weight Initializations, Data Orders, and Early Stopping. *arXiv preprint arXiv:2002.06305*, 2020.

[8] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In *NeurIPS*, 2020.

[9] Ethan Perez, Douwe Kiela, and Kyunghyun Cho. True Few-Shot Learning with Language Models. *arXiv preprint arXiv:2105.11447*, 2021. URL `http://arxiv.org/abs/2105.11447`.

[10] Xiang Lisa Li and Percy Liang. Prefix-tuning: Optimizing continuous prompts for generation. *ACL*, 2021.

[11] Karen Hambardzumyan, Hrant Khachatrian, and Jonathan May. Warp: Word-level adversarial reprogramming. *ACL*, 2021.

[12] Brian Lester, Rami Al-Rfou, and Noah Constant. The power of scale for parameter-efficient prompt tuning. *EMNLP*, 2021.

[13] Chunyuan Li, Heerad Farkhoor, Rosanne Liu, and Jason Yosinski. Measuring the intrinsic dimension of objective landscapes. In *ICLR*, 2018.

[14] Armen Aghajanyan, Luke Zettlemoyer, and Sonal Gupta. Intrinsic dimensionality explains the effectiveness of language model fine-tuning. *ACL*, 2021.

[15] Sylvestre-Alvise Rebuffi, Hakan Bilen, and Andrea Vedaldi. Efficient parametrization of multi-domain deep neural networks. In *CVPR*, 2018.

[16] Zhaojiang Lin, Andrea Madotto, and Pascale Fung. Exploring versatile generative language model via parameter-efficient transfer learning. In *EMNLP Findings*, 2020.

[17] Aston Zhang, Yi Tay, SHUAI Zhang, Alvin Chan, Anh Tuan Luu, Siu Hui, and Jie Fu. Beyond fully-connected layers with quaternions: Parameterization of hypercomplex multiplications with 1/n parameters. In *ICLR*, 2021.

[18] Alex Wang, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel R. Bowman. GLUE: A multi-task benchmark and analysis platform for natural language understanding. In *ICLR*, 2019.

[19] Alex Wang, Yada Pruksachatkun, Nikita Nangia, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel R Bowman. Superglue: a stickier benchmark for general-purpose language understanding systems. In *NeurIPS*, 2019.

[20] Dan Hendrycks and Kevin Gimpel. Gaussian error linear units (gelus). *arXiv preprint arXiv:1606.08415*, 2016.

[21] Tianyi Zhang, Felix Wu, Arzoo Katiyar, Kilian Q Weinberger, and Yoav Artzi. Revisiting Few-sample BERT Fine-tuning. In *ICLR*, 2021.

[22] Hyung Won Chung, Thibault Févry, Henry Tsai, Melvin Johnson, and Sebastian Ruder. Rethinking Embedding Coupling in Pre-trained Language Models. In *ICLR*, 2021.

[23] Andreas Rücklé, Gregor Geigle, Max Glockner, Tilman Beck, Jonas Pfeiffer, Nils Reimers, and Iryna Gurevych. AdapterDrop: On the Efficiency of Adapters in Transformers. *EMNLP*, 2021.

[24] Sanjeev Arora, Nadav Cohen, and Elad Hazan. On the optimization of deep networks: Implicit acceleration by overparameterization. In *ICML*, 2018.

[25] Yeming Wen, Dustin Tran, and Jimmy Ba. BatchEnsemble: An Alternative Approach to Efficient Ensemble and Lifelong Learning. In *ICLR*, 2020.

[26] Hector Levesque, Ernest Davis, and Leora Morgenstern. The winograd schema challenge. In *KR*, 2012.

[27] Tianyi Zhang, Felix Wu, Arzoo Katiyar, Kilian Q Weinberger, and Yoav Artzi. Revisiting few-sample bert fine-tuning. In *ICLR*, 2021.

[28] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. Transformers: State-of-the-art natural language processing. In *EMNLP: System Demonstrations*, 2020.

[29] Tuan Le, Marco Bertolini, Frank Noé, and Djork-Arné Clevert. Parameterized hypercomplex graph neural networks for graph classification. *ICANN*, 2021.

[30] Rabeeh Karimi Mahabadi, Sebastian Ruder, Mostafa Dehghani, and James Henderson. Parameter-efficient multi-task fine-tuning for transformers via shared hypernetworks. In *ACL*, 2021.

[31] Jonas Pfeiffer, Aishwarya Kamath, Andreas RückĹe, Cho Kyunghyun, and Iryna Gurevych. AdapterFusion: Non-destructive task composition for transfer learning. In *EACL*, 2021.

[32] Quoc Le, Tamás Sarlós, and Alex Smola. Fastfood-approximating kernel expansions in loglinear time. In *ICML*, 2013.

[33] Han Cai, Chuang Gan, Ligeng Zhu, and Song Han. Tinytl: Reduce memory, not parameters for efficient on-device learning. *NeurIPS*, 2020.

[34] Shauli Ravfogel, Elad Ben-Zaken, and Yoav Goldberg. Bitfit: Simple parameter-efficient fine-tuning for transformer-based masked languagemodels. *arXiv preprint arXiv:2106.10199*.

[35] Andrew Brock, Soham De, Samuel L Smith, and Karen Simonyan. High-performance large-scale image recognition without normalization. *ICML*, 2021.

[36] Sinong Wang, Belinda Li, Madian Khabsa, Han Fang, and Hao Ma. Linformer: Self-attention with linear complexity. *arXiv preprint arXiv:2006.04768*, 2020.

[37] Ahmet Üstün, Arianna Bisazza, Gosse Bouma, and Gertjan van Noord. Udapter: Language adaptation for truly universal dependency parsing. In *EMNLP*, 2020.

[38] Emmanouil Antonios Platanios, Mrinmaya Sachan, Graham Neubig, and Tom Mitchell. Contextual parameter generation for universal neural machine translation. In *EMNLP*, 2018.

[39] Jonathan Pilault, Amine El hattami, and Christopher Pal. Conditionally adaptive multi-task learning: Improving transfer learning in NLP using fewer parameters & less data. In *ICLR*, 2021.

[40] Chase J Gaudet and Anthony S Maida. Deep quaternion networks. In *IJCNN*, 2018.

[41] Titouan Parcollet, Ying Zhang, Mohamed Morchid, Chiheb Trabelsi, Georges Linarès, Renato de Mori, and Yoshua Bengio. Quaternion convolutional neural networks for end-to-end automatic speech recognition. In *Interspeech*, 2018.

[42] Titouan Parcollet, Mirco Ravanelli, Mohamed Morchid, Georges Linarès, Chiheb Trabelsi, Renato De Mori, and Yoshua Bengio. Quaternion recurrent neural networks. In *ICLR*, 2018.

[43] Xuanyu Zhu, Yi Xu, Hongteng Xu, and Changjian Chen. Quaternion convolutional neural networks. In *ECCV*, 2018.

[44] Yi Tay, Aston Zhang, Anh Tuan Luu, Jinfeng Rao, Shuai Zhang, Shuohang Wang, Jie Fu, and Siu Cheung Hui. Lightweight and efficient neural natural language processing with quaternion networks. In *ACL*, 2019.

[45] Tianlong Chen, Jonathan Frankle, Shiyu Chang, Sijia Liu, Yang Zhang, Zhangyang Wang, and Michael Carbin. The lottery ticket hypothesis for pre-trained bert networks. *NeurIPS*, 2020.

[46] Sai Prasanna, Anna Rogers, and Anna Rumshisky. When BERT Plays the Lottery, All Tickets Are Winning. In *EMNLP*, 2020.

[47] Shrey Desai, Hongyuan Zhan, and Ahmed Aly. Evaluating lottery tickets under distributional shifts. In *DeepLo*, 2019.

[48] Davis Blalock, Jose Javier Gonzalez Ortiz, Jonathan Frankle, and John Guttag. What is the state of neural network pruning? *arXiv preprint arXiv:2003.03033*, 2020.

[49] Alex Warstadt, Amanpreet Singh, and Samuel R. Bowman. Neural network acceptability judgments. *TACL*, 2019.

[50] Richard Socher, Alex Perelygin, Jean Wu, Jason Chuang, Christopher D. Manning, Andrew Ng, and Christopher Potts. Recursive deep models for semantic compositionality over a sentiment treebank. In *EMNLP*, 2013.

[51] William B. Dolan and Chris Brockett. Automatically constructing a corpus of sentential paraphrases. In *IWP*, 2005.

[52] Daniel Cer, Mona Diab, Eneko Agirre, Iñigo Lopez-Gazpio, and Lucia Specia. SemEval-2017 task 1: Semantic textual similarity multilingual and crosslingual focused evaluation. In *SemEval*, 2017.

[53] Adina Williams, Nikita Nangia, and Samuel Bowman. A broad-coverage challenge corpus for sentence understanding through inference. In *NAACL*, 2018.

[54] Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. SQuAD: 100,000+ questions for machine comprehension of text. In *EMNLP*, 2016.

[55] Ido Dagan, Oren Glickman, and Bernardo Magnini. The pascal recognising textual entailment challenge. In *Machine Learning Challenges Workshop*, 2005.

[56] Roy Bar-Haim, Ido Dagan, Bill Dolan, Lisa Ferro, and Danilo Giampiccolo. The second pascal recognising textual entailment challenge. *Second PASCAL Challenges Workshop on Recognising Textual Entailment*, 2006.

[57] Danilo Giampiccolo, Bernardo Magnini, Ido Dagan, and Bill Dolan. The third PASCAL recognizing textual entailment challenge. In *ACL-PASCAL Workshop on Textual Entailment and Paraphrasing*, 2007.

[58] Luisa Bentivogli, Ido Dagan, Hoa Trang Dang, Danilo Giampiccolo, and Bernardo Magnini. The fifth pascal recognizing textual entailment challenge. In *TAC*, 2009.

[59] Melissa Roemmele, Cosmin Adrian Bejan, and Andrew S Gordon. Choice of plausible alternatives: An evaluation of commonsense causal reasoning. In *AAAI Spring Symposium Series*, 2011.

[60] Marie-Catherine De Marneffe, Mandy Simons, and Judith Tonhauser. The commitmentbank: Investigating projection in naturally occurring discourse. In *proceedings of Sinn und Bedeutung*, 2019.

[61] Daniel Khashabi, Snigdha Chaturvedi, Michael Roth, Shyam Upadhyay, and Dan Roth. Looking beyond the surface: A challenge set for reading comprehension over multiple sentences. In *NAACL*, 2018.

[62] Sheng Zhang, Xiaodong Liu, Jingjing Liu, Jianfeng Gao, Kevin Duh, and Benjamin Van Durme. Record: Bridging the gap between human and machine commonsense reading comprehension. *arXiv preprint arXiv:1810.12885*, 2018.

[63] Christopher Clark, Kenton Lee, Ming-Wei Chang, Tom Kwiatkowski, Michael Collins, and Kristina Toutanova. BoolQ: Exploring the surprising difficulty of natural yes/no questions. In *NAACL*, 2019.

[64] Mohammad Taher Pilehvar and Jose Camacho-Collados. Wic: the word-in-context dataset for evaluating context-sensitive meaning representations. In *NAACL*, 2019.

[65] Karin Kipper Schuler. Verbnet: A broad-coverage, comprehensive verb lexicon. *PhD Thesis*, 2005.

[66] George A Miller. Wordnet: a lexical database for english. *Communications of the ACM*, 1995.

[67] Thomas Wolf, Quentin Lhoest, Patrick von Platen, Yacine Jernite, Mariama Drame, Julien Plu, Julien Chaumond, Clement Delangue, Clara Ma, Abhishek Thakur, Suraj Patil, Joe Davison, Teven Le Scao, Victor Sanh, Canwen Xu, Nicolas Patry, Angie McMillan-Major, Simon Brandeis, Sylvain Gugger, François Lagunas, Lysandre Debut, Morgan Funtowicz, Anthony Moi, Sasha Rush, Philipp Schmidd, Pierric Cistac, Victor Muštar, Jeff Boudier, and Anna Tordjmann. Datasets. *GitHub. Note: https://github.com/huggingface/datasets*, 2020.

Table 4: Selected learning rates for all methods.

Method	Learning rate
$T5_{BASE}$	3e−4
ADAPTER	3e−4
PFEIFFER-ADAPTER	3e−4
ADAPTERDROP	3e−4
ADAPTER-LOWRANK	3e−3
PROMPT TUNING-R	3e−2
PROMPT TUNING-T	3e−1
INTRINSIC-SAID	3e−2
BITFIT	3e−4
PHM-ADAPTER	3e−3
COMPACTER	3e−3
COMPACTER++	3e−3

Table 5: Selected learning rates for all methods, when we also fine-tune the output layer.

Method	Learning rate
ADAPTER	3e−3
PFEIFFER-ADAPTER	3e−4
ADAPTERDROP	3e−4
ADAPTER-LOWRANK	3e−3
BITFIT	3e−4
PHM-ADAPTER	3e−3
COMPACTER	3e−3
COMPACTER++	3e−3

A Experimental Details

Datasets We run all experiments on the standard GLUE benchmark [18] with Creative Commons license (CC BY 4.0) and the SUPERGLUE benchmark [19]. These benchmark consist of multiple datasets: CoLA [49], SST-2 [50], MRPC [51], QQP[9], STS-B [52], MNLI [53], QNLI [54], and RTE, which is a combination of data from RTE1 [55], RTE2 [56], RTE3 [57], RTE5 [58], COPA [59], CB [60], MultiRC [61], ReCoRD [62], BoolQ [63], and WiC [64] where sentences are selected from VerbNet [65], WordNet [66], and Wiktionary. We download all datasets from the HuggingFace Datasets library [67].

Low-resource fine-tuning For the experiment conducted in §5.6, we set the number of epochs to 1000, 200, 100, 50, 25, for datasets subsampled to size 100, 500, 1000, 2000, and 4000 respectively. Based on our results, this is sufficient to allow the models to converge. We save a checkpoint every 250 steps for all models and report the results for the hyper-parameters performing the best on the validation set for each task.

Data pre-processing: Following Raffel et al. [3], we cast all datasets into a sequence-to-sequence format. We recast STS-B as a 21-class classification task by rounding its target scores to their nearest increment of 0.2.

Computing infrastructure: We run the experiments in Table 1, 2, 9, and 3 on one NVIDIA GEFORCE RTX 3090, and experiments in §5.6 on one GEFORCE GTX 1080 TI GPU.

Training hyper-parameters: For the experiments on GLUE, we set the maximum sequence length to 128 and batch size to 100. Following Raffel et al. [3], we use maximum sequence length of 256 for the tasks in SUPERGLUE, and for ReCoRD, we set it to 512. We used batch size of 32 for SUPERGLUE, and for ReCoRD, we set it to 16 due to the GPU memory limit. For results in §5.6, we set the batch size to 40 to match the lower GPU memory of GEFORCE GTX 1080 TI GPU. For setting the learning rates, we trained all methods with $3e{-}5$, $3e{-}4$, $3e{-}3$, $3e{-}2$, and $3e{-}1$ and use the learning rate performing the best on the validation set for each method. Table 4 shows the final selected learning rate for each method reported in Table 1. For the method variants where we also fine-tune the final output layer (Table 7), we report the selected learning rate in Table 5. We train all models with the AdamW optimizer from the HuggingFace library [28] with default hyper-parameters of $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\epsilon = 1e{-}8$. We set warm-up steps to 500 for all methods in Table 1 and 7. We set the warm-up steps to 0 for all methods in Table 2 and 9, which based on our experiments, improved the results for all methods.

[9] https://data.quora.com/First-Quora-Dataset-Release-Question-Pairs

Table 6: Performance of all methods on the tasks in GLUE for different values of hyper-parameters. For each method, we report the total number of parameters across all tasks and the number of parameters that are trained for each task as a multiple and proportion of $T5_{BASE}$ model [3]. For MNLI, we report accuracy on the matched validation set. For MRPC and QQP, we report accuracy and F1. For STS-B, we report Pearson and Spearman correlation coefficients. For CoLA, we report Matthews correlation. For all other tasks, we report accuracy. Bold fonts indicate the best results in each block.

Method	#Total params	Trained params / per task	CoLA	SST-2	MRPC	QQP	STS-B	MNLI	QNLI	RTE	Avg
INTRINSIC-SAID ($d'=0.4$K)	1.001	0.0002%	0.0	92.55	78.43/85.62	90.25/87.19	90.43/90.66	69.93	89.31	58.99	75.76
INTRINSIC-SAID ($d'=1.4$K)	1.001	0.0006%	52.40	93.35	**89.22/92.41**	90.44/87.31	89.86/90.23	82.01	93.12	67.63	84.36
INTRINSIC-SAID ($d'=2.5$K)	1.001	0.0011%	45.78	93.92	89.22/92.20	**90.43/87.37**	90.32/90.90	82.86	93.17	64.03	83.65
INTRINSIC-SAID ($d'=10$K)	1.001	0.0045%	56.13	93.58	88.73/91.99	90.34/87.18	**90.63/90.99**	84.84	93.36	**71.22**	85.36
INTRINSIC-SAID ($d'=20$K)	1.001	0.0090%	**58.69**	**94.15**	88.24/91.78	90.28/87.13	90.06/90.45	**85.23**	**93.39**	70.50	**85.45**
PHM-ADAPTER ($n=4$)	1.018	0.239%	59.21	93.69	87.25/90.91	90.23/86.99	90.55/90.73	85.93	**93.04**	69.78	85.30
PHM-ADAPTER ($n=8$)	1.011	0.160%	**61.84**	93.58	91.18/93.57	**90.25/87.08**	**90.74/91.07**	85.74	92.93	70.50	86.23
PHM-ADAPTER ($n=12$)	1.013	0.179%	57.35	**94.50**	**91.67/93.86**	90.25/87.05	90.45/90.84	**85.97**	92.92	**75.54**	**86.40**
COMPACTER ($n=4$)	1.004	0.073%	**63.75**	93.00	89.22/92.31	**90.23/87.03**	90.31/90.74	85.61	92.88	**77.70**	**86.62**
COMPACTER ($n=8$)	1.004	0.073%	61.78	**93.81**	90.20/93.10	**90.23/87.03**	90.16/90.44	**85.78**	93.08	74.10	86.34
COMPACTER ($n=12$)	1.004	0.073%	61.38	93.69	**91.18/93.71**	90.11/86.88	**90.53/90.98**	85.76	**93.12**	70.50	86.17
COMPACTER++ ($n=4$)	1.002	0.047%	61.27	93.81	90.69/93.33	90.17/86.93	**90.46/90.93**	**85.71**	**93.08**	**74.82**	**86.47**
COMPACTER++ ($n=8$)	1.002	0.047%	62.79	92.55	88.24/91.95	90.16/86.94	90.43/90.78	85.36	92.82	73.38	85.95
COMPACTER++ ($n=12$)	1.002	0.048%	**63.01**	**93.92**	**91.18/93.75**	**90.23/87.01**	90.40/90.65	85.46	92.88	71.22	86.34

B Impact of Hyper-parameters

In this section, we study the impact of hyper-parameters for each method reported in Table 1. We report the results in Table 6.

Impact of dimension (d') on INTRINSIC-SAID Increasing the dimension d' for INTRINSIC-SAID method often improves results. Though, as discussed in [14], d' is task-dependent so needs to be tuned for every new dataset to achieve optimal performance.

Impact of n on PHM-ADAPTER: Table 6 shows the results for varying values of $n=\{4,8,12\}$. We experiment with adapters of bottleneck size $d\in\{24,48,96\}$.

For the $T5_{BASE}$ model with $k=768$, the condition $kd>n^4$ discussed in §4 is partially satisfied for $d=24$ and $n=4,8$ and fully satisfied for $d\in\{48,96\}$ and $n=4,8,12$. Note that this condition is satisfied for larger versions of the T5 model, i.e., T5-large (770 million parameters, $k=1024$), T5-3B (2.8 billion parameters, $k=1024$), and T5-11B (11 billion parameters, $k=1024$) with adapter hidden size $d\in\{24,32,48,96\}$ and $n=2,4,8,12$. Due to the huge computational costs of training these models, we could not run experiments on such a large scale. Nevertheless, we observe substantial parameter reduction using PHM-ADAPTER.

In Table 6, we report the number of parameters for $d=24$ for all methods. Compared to ADAPTER, PHM-ADAPTER with $n=8$ reduces the parameters substantially by $5.2\times$.

Impact of n on COMPACTER: For COMPACTER and COMPACTER++, we observe that the number of trainable parameters is almost constant across different values of n. This is due to the fact that the number of trainable parameters in layernorms (LN) and biases (B) in each LPHM layer make up a high proportion of parameters for our methods. For instance for $n=4$, for COMPACTER with 0.073% of trainable parameters, LN and B make up 28.49% and 23.51% respectively of its trainable parameters; for COMPACTER++ with 0.047% of trainable parameters, LN and B make up 44.01% and 18.15% respectively of its parameters; while for PHM-ADAPTER with 0.239% of trainable parameters, LN and B make up only 8.63% and 7.12% respectively of its parameters. Consequently, simply removing biases from adapters, and exploring ideas of training language models without layer normalizations [35] can be promising directions on reducing parameters further, which we leave to future work.

COMPACTER has more than an order of magnitude fewer parameters compared to ADAPTER, with a parameter reduction at a remarkable rate of $11.4\times$. COMPACTER++ even reduces the parameters further by $17.7\times$ in total.

C Results with Fine-tuning the Output Layer

Table 7 shows the results for the methods in Table 1 with fine-tuning the output layer. The parameters of the output layer dominate the parameters of each method and thus reduce the relative parameter savings. The standard adapter obtains the largest improvement in performance when fine-tuning the output layer compared to the results in Table 1. In contrast, our proposed methods perform well with or without fine-tuning the output layer.

Table 7: Performance of all methods on the tasks in GLUE, where the output layer is tuned. For each method, we report the total number of parameters across all tasks and the percentage of parameters that are trained for each task as a multiple and proportion of T5_{BASE} model [3]. For MNLI, we report accuracy on the matched validation set. For MRPC and QQP, we report accuracy and F1. For STS-B, we report Pearson and Spearman correlation coefficients. For CoLA, we report Matthews correlation. For all other tasks, we report accuracy. Bold fonts indicate the best results in each block.

Method	#Total params	Trained params / per task	CoLA	SST-2	MRPC	QQP	STS-B	MNLI	QNLI	RTE	Avg
Baselines											
ADAPTER	1.065	11.89%	61.80	**94.15**	88.24/91.67	**90.27/87.05**	**91.51/91.71**	86.02	92.64	**76.26**	**86.48**
PFEIFFER-ADAPTER	1.032	11.49%	**64.76**	93.58	87.75/91.58	90.16/87.17	91.21/91.50	86.16	**93.30**	73.38	86.41
ADAPTERDROP	1.038	11.56%	61.67	93.69	84.80/89.20	90.14/87.17	90.92/91.34	**86.24**	93.23	73.38	85.62
ADAPTER-LOWRANK	1.004	11.13%	62.82	93.81	88.73/91.99	90.34/87.19	90.51/90.58	85.81	92.93	74.82	86.32
BITFIT	1.010	11.19%	57.13	**94.15**	**89.71/92.78**	90.07/87.02	90.91/91.22	85.34	93.06	68.35	85.43
Our Proposed Methods											
PHM-ADAPTER ($n=4$)	1.017	11.30%	62.79	93.58	**89.22/92.41**	90.23/87.01	90.61/90.81	**86.06**	92.95	**75.54**	**86.47**
PHM-ADAPTER ($n=8$)	1.011	11.22%	61.24	**94.38**	88.73/91.99	90.28/87.08	90.53/90.98	85.94	**93.03**	73.38	86.14
PHM-ADAPTER ($n=12$)	1.013	11.24%	**65.25**	93.69	88.73/92.04	**90.34/87.16**	**90.75/90.89**	85.74	92.92	72.66	86.38
COMPACTER ($n=4$)	1.004	11.13%	**61.27**	93.58	88.24/91.67	90.25/87.08	**90.67/91.02**	**85.82**	92.92	73.38	85.99
COMPACTER ($n=8$)	1.004	11.13%	60.31	**93.81**	89.71/92.63	90.23/87.02	90.49/90.85	85.19	**93.08**	71.94	85.93
COMPACTER ($n=12$)	1.004	11.13%	59.25	93.12	**91.18/93.75**	**90.31/87.16**	90.37/90.82	85.33	92.97	**75.54**	**86.35**
COMPACTER++ ($n=4$)	1.002	11.11%	**64.28**	**94.27**	90.20/92.96	**90.23/87.04**	**90.27/90.61**	**85.80**	**92.97**	73.38	**86.55**
COMPACTER++ ($n=8$)	1.002	11.11%	63.78	93.58	**90.20/93.01**	90.19/87.02	90.12/90.56	85.57	92.84	70.50	86.12
COMPACTER++ ($n=12$)	1.002	11.11%	62.05	93.23	87.75/91.58	90.19/86.97	90.08/90.48	85.52	92.75	**79.86**	86.41

D Results on SUPERGLUE

Table 8 shows the performance of our proposed methods on SUPERGLUE for different values of n. We include the learning rate obtaining the best validation performance for all methods reported in Table 2 in Table 11.

Table 8: Performance of our proposed methods on the tasks in SUPERGLUE for different values of n. For each method, we report the total number of parameters across all tasks and the percentage of parameters that are trained for each task as a multiple and proportion of T5_{BASE} model [3]. For CB, we report accuracy and F1. For MultiRC, we report F1 over all answer-options (F1_a) and exact match of each question's set of answers (EM) [19]. For ReCoRD, we report F1 and EM scores. For all other tasks, we report accuracy. Bold fonts indicate the best results in each block.

Method	#Total params	Trained params / per task	BoolQ	CB	COPA	MultiRC	ReCoRD	WiC	Avg
PHM-ADAPTER ($n=4$)	1.013	0.24%	**80.31**	**85.71/73.52**	44.0	**71.99/51.65**	**74.62/73.60**	67.40	**69.20**
PHM-ADAPTER ($n=8$)	1.008	0.160%	79.39	82.14/69.87	44.0	71.49/50.77	74.46/73.48	67.71	68.15
PHM-ADAPTER ($n=12$)	1.009	0.179%	79.33	78.57/75.43	**52.0**	70.48/50.66	74.14/73.14	**68.65**	69.16
COMPACTER ($n=4$)	1.003	0.073%	**79.88**	89.29/82.51	42.0	**71.87/51.98**	**74.64/73.59**	65.83	70.18
COMPACTER ($n=8$)	1.003	0.073%	79.57	85.71/80.06	**56.0**	70.75/49.67	74.56/73.57	**70.85**	71.19
COMPACTER ($n=12$)	1.003	0.073%	78.59	**96.43/87.44**	48.0	70.80/49.67	74.49/73.54	65.20	**71.57**
COMPACTER++ ($n=4$)	1.002	0.047%	**79.94**	85.71/80.06	50.0	72.16/50.33	74.63/73.60	**68.34**	70.53
COMPACTER++ ($n=8$)	1.002	0.047%	78.23	82.14/70.87	48.0	**71.61/51.43**	**74.62/73.64**	67.71	68.69
COMPACTER++ ($n=12$)	1.002	0.048%	78.84	**92.86/84.96**	**52.0**	70.68/50.99	74.55/73.50	68.03	**71.82**

E Impact of Model Size

Table 9 shows the results of methods using $T5_{SMALL}$ (60M parameters) on GLUE. For all adapter-based methods, we experiment with adapters of bottleneck size of $\{16, 32, 64\}$. For our methods, we experiment with $n = \{4,8,16\}$.

All parameter-efficient fine-tuning methods are performing worse than full fine-tuning with this small model size. This is in contrast to the results of Table 1 and 2, where some parameter-efficient fine-tuning methods were able to perform on par or outperform full fine-tuning with the larger model size of $T5_{BASE}$ (222M parameters). Among all methods, adapters, and our proposed methods perform the best. We report the learning rate performing the best on the validation set of each method in Table 10.

Table 9: Performance of all methods on the tasks in GLUE. For each method, we report the total number of parameters across all tasks and the percentage of parameters that are trained for each task as a multiple and proportion of $T5_{SMALL}$ model [3]. For MNLI, we report accuracy on the matched validation set. For MRPC and QQP, we report accuracy and F1. For STS-B, we report Pearson and Spearman correlation coefficients. For CoLA, we report Matthews correlation. For all other tasks, we report accuracy. Bold fonts indicate the best results in each block. We repeat the experiments marked with $*$ multiple times for different seeds, but they were not successful.

Method	#Total params	Trained params / per task	CoLA	SST-2	MRPC	QQP	STS-B	MNLI	QNLI	RTE	Avg
Baselines											
$T5_{SMALL}$	8×1	100%	**46.90**	**91.74**	87.25/90.97	**90.07/86.68**	88.75/89.20	**82.20**	**90.59**	**65.47**	**82.71**
ADAPTER	1.054	0.698%	36.88	90.83	**88.73/91.93**	88.09/84.06	88.98/89.34	80.50	89.75	62.59	81.06
ADAPTERDROP	1.009	0.139%	34.73	89.91	83.33/88.36	87.96/83.89	88.73/88.80	79.33	89.86	61.87	79.71
PFEIFFER-ADAPTER	1.027	0.363%	38.86	90.48	85.78/89.90	87.82/84.26	**89.24/89.56**	80.63	89.84	57.55	80.36
ADAPTER-LOWRANK	1.005	0.090%	40.55	90.60	84.80/89.20	88.01/83.98	88.04/88.27	79.92	89.95	61.15	80.41
PROMPT TUNING-R	1.007	0.085%	0.0*	86.35	68.14/81.05	87.48/83.91	87.35/87.87	76.27	88.49	50.36	72.48
PROMPT TUNING-T	1.007	0.085%	0.0*	79.59	71.08/82.18	87.76/83.55	87.48/87.76	74.65	89.02	57.55	72.78
BITFIT	1.015	0.190%	25.59	90.48	84.80/89.42	88.01/83.77	87.58/87.89	78.15	88.94	63.31	78.90
INTRINSIC-SAID	1.003	0.033%	0.0*	90.25	84.80/89.05	88.07/84.00	87.81/88.08	79.02	89.90	52.52	75.77
Our Proposed Methods											
PHM-ADAPTER ($n=4$)	1.015	0.216%	40.08	90.60	**86.27/90.21**	**88.26/84.25**	**89.56/89.88**	**80.73**	**90.10**	60.43	80.94
PHM-ADAPTER ($n=8$)	1.011	0.170%	37.85	90.48	82.84/87.72	88.08/84.07	89.07/89.46	80.68	89.64	61.87	80.16
PHM-ADAPTER ($n=16$)	1.031	0.414%	36.27	**90.83**	83.82/88.34	88.03/84.02	87.94/88.44	80.04	89.95	58.99	79.70
COMPACTER ($n=4$)	1.005	0.090%	**44.65**	89.45	84.80/89.20	88.00/83.96	88.19/88.47	79.54	89.66	64.03	80.90
COMPACTER ($n=8$)	1.005	0.091%	42.90	89.56	84.31/89.12	88.01/83.95	88.51/88.79	79.60	89.68	**66.19**	**80.97**
COMPACTER ($n=16$)	1.006	0.097%	40.12	89.22	85.29/89.86	88.08/84.06	89.28/89.60	79.87	89.71	59.71	80.44
COMPACTER++ ($n=4$)	1.003	0.059%	39.89	90.37	84.31/89.26	88.04/83.99	88.69/88.98	79.45	89.05	63.31	80.49
COMPACTER++ ($n=8$)	1.003	0.059%	34.98	90.37	83.82/88.50	88.02/83.99	88.87/89.30	79.39	89.57	64.03	80.08
COMPACTER++ ($n=16$)	1.003	0.065%	37.54	89.79	85.78/89.90	88.01/83.96	88.93/89.30	79.35	89.40	64.75	80.61

Table 10: Selected learning rates for all methods with $T5_{SMALL}$

Method	Learning rate
$T5_{SMALL}$	3e−4
ADAPTER	3e−3
PFEIFFER-ADAPTER	3e−4
ADAPTERDROP	3e−3
ADAPTER-LOWRANK	3e−3
PROMPT TUNING-R	3e−2
PROMPT TUNING-T	3e−1
INTRINSIC-SAID	3e−2
BITFIT	3e−3
PHM-ADAPTER	3e−3
COMPACTER	3e−3
COMPACTER++	3e−3

Table 11: Selected learning rates for all methods with $T5_{BASE}$ on SUPERGLUE.

Method	Learning rate
$T5_{BASE}$	3e−4
ADAPTER	3e−4
PFEIFFER-ADAPTER	3e−4
ADAPTERDROP	3e−4
ADAPTER-LOWRANK	3e−3
PROMPT TUNING-R	3e−2
PROMPT TUNING-T	3e−1
BITFIT	3e−4
INTRINSIC-SAID	3e−3
PHM-ADAPTER	3e−3
COMPACTER	3e−3
COMPACTER++	3e−3