Hadamard Adapter: An Extreme Parameter-Efficient Adapter Tuning Method for Pre-trained Language Models

Yuyan Chen*
chenyuyan21@m.fudan.edu.cn
Shanghai Key Laboratory of Data Science, School of Computer Science, Fudan University
Shanghai, China

Qiang Fu†
qifu@microsoft.com
Microsoft
Beijing, China

Ge Fan
ge.fan@outlook.com
Tencent
Shenzhen, China

Lun Du
ludu@microsoft.com
Microsoft
Beijing, China

Jian-Guang Lou
jlou@microsoft.com
Microsoft
Beijing, China

Shi Han
shihan@microsoft.com
Microsoft
Beijing, China

Dongmei Zhang
dongmeiz@microsoft.com
Microsoft
Beijing, China

Zhixu Li†
zhixuli@fudan.edu.cn
Shanghai Key Laboratory of Data Science, School of Computer Science, Fudan University
Shanghai, China

Yanghua Xiao†
shawyh@fudan.edu.cn
Shanghai Key Laboratory of Data Science, School of Computer Science, Fudan University, Fudan-Aishu Cognitive Intelligence Joint Research Center
Shanghai, China

ABSTRACT

Recent years, Pre-trained Language models (PLMs) have swept into various fields of artificial intelligence and achieved great success. However, most PLMs, such as T5 and GPT3, have a huge amount of parameters, fine-tuning them is often expensive and time consuming, and storing them takes up a lot of space. Therefore, it is necessary to adopt a parameter-efficient approach to reduce parameters of PLMs in fine-tuning without compromising their performance in downstream tasks. In this paper, we design a novel adapter which only acts on self-attention outputs in PLMs. This adapter adopts element-wise linear transformation using Hadamard product, hence named as Hadamard adapter, requires the fewest parameters compared to previous parameter-efficient adapters. In addition, we also summarize some tuning patterns for Hadamard adapter shared by various downstream tasks, expecting to provide some guidance for further parameter reduction with shared adapters in future studies. The experiments conducted on the widely-used GLUE benchmark with several SOTA PLMs prove that the Hadamard adapter achieves competitive performance with only 0.033% parameters compared with full fine-tuning, and it has the fewest parameters compared with other adapters. Moreover, we further find that there is also some redundant layers in the Hadamard adapter which can be removed to achieve more parameter efficiency with only 0.022% parameters.

CCS CONCEPTS

• **Computing methodologies** → **Natural language processing**.

KEYWORDS

Adapter Tuning, Parameter-Efficiency, Pre-trained Language Models

ACM Reference Format:
Yuyan Chen, Qiang Fu, Ge Fan, Lun Du, Jian-Guang Lou, Shi Han, Dongmei Zhang, Zhixu Li, and Yanghua Xiao. 2023. Hadamard Adapter: An Extreme Parameter-Efficient Adapter Tuning Method for Pre-trained Language Models. In *Proceedings of the 32nd ACM International Conference on Information and Knowledge Management (CIKM '23), October 21–25, 2023, Birmingham, United Kingdom.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3583780.3614904

*Work done while this author was an intern at Microsoft Research.
†The corresponding authors.

CIKM '23, October 21–25, 2023, Birmingham, United Kingdom

ACM ISBN 979-8-4007-0124-5/23/10…$15.00
https://doi.org/10.1145/3583780.3614904

1 INTRODUCTION

Recent years, Pre-trained Language models (PLMs) have swept into various fields of artificial intelligence and achieved great success. The mainstream paradigm for adapting PLMs to downstream tasks is fine-tuning. As most PLMs, such as T5 [31], GPT3 [4], have a large amount of parameters, fine-tuning them is often expensive and time consuming, and storing them takes up a lot of space. It has

been revealed that there are a lot of redundant parameters in the process of fine-tuning [5, 7]. Thus, it is necessary to greatly reduce the scale of parameters in fine-tuning without compromising PLMs' performance in downstream tasks.

Previous parameter-efficient fine-tuning of PLMs mainly contains three categories of methods, i.e., adapter tuning, prefix tuning, and prompt tuning. Adapter tuning [15] is to inject a small neural network module into each or some layers of the PLMs. During fine-tuning, only the parameters of this small module need to be learned. It has promising performance in NLP, which achieves comparable performance with fine-tuning while adding no more than 4% task-specific parameters [15, 21]. Prefix tuning [20] and prompt tuning [18] preset additional adjustable prefix tokens in the input or hidden layer, and only these soft prompts are trained during the fine-tuning of downstream tasks. In addition to the above three parameter-efficient fine-tuning ways, the existing efforts also works on model compression [5], including knowledge distillation [2, 14], which transfers the knowledge learned by large models to small models, such that small models can have the generalization ability of large models; quantization [9, 39], which reduces the accuracy of large models within the acceptable range; pruning [11, 33], which removes less useful connections in the model; and structure optimization, such as matrix decomposition [35], parameter sharing [36], etc. However, although the current endeavors achieve competitive performance in downstream tasks with much fewer parameters, we believe there is still room for improvement in parametric efficiency.

It's well-known that the attention mechanism, especially self-atten-tion, is one of the core modules that enable PLMs achieve superior performance in various downstream tasks [37]. Thus, a possible way to significantly reduce the scale of parameters for fine-tuning might be designing an adapter to work with the self-attention module in PLMs. We also find related research on adapter tuning that injects adapters into the self-attention layer, such as IA3 [22]. There are three questions need to be answered when designing the adapter. **Q1.** *Where should the adapter that acts on self-attention outputs be injected into the PLMs?* **Q2.** *What is the suitable form of the adapter that satisfies both competitive performance and parameter-efficiency?* **Q3.** *What other essential parameters should not be frozen in adapter tuning?* To answer these questions, we conduct the following empirical studies: i) Analyzing the changes of self-attention outputs before and after full fine-tuning to verify the importance of self-attention which is therefore necessary to inject the adapter; ii) Comparing the difference among all fitting functions to select the suitable form of the adapter; iii) Analyzing the gradients of PLMs after fine-tuning on downstream tasks to select out the modules of great importance which should to be trained in the adapter tuning.

According to the empirical analysis, we propose a novel adapter tuning method as follows: We first learn the classification module to output prediction results on a given downstream task, without updating the PLMs' other parameters. Since the classification module is a linear model, this step requires light-weight computation cost. Then we design an adapter and inject it right after the multi-head self-attention outputs of PLMs. Particularly, we freeze all parameters except parameters in the designed adapter and the subsequent normalization module for continuous fine-tuning. As there are usually multiple layers with the same architecture in PLMs, e.g., BERT [8] model of base version has 12 layers, we inject such an adapter module in each layer of PLMs. In designing the adapter, we only adopt element-wise linear transformation, rather than high-order ones, as the computational logic for the adapter. Specifically, the adapter includes a weight vector and a bias vector which have the same dimension as the output of the multi-head self-attention module. The multi-head self-attention output is multiplied by the weight vector of the adapter using the element-wise product (also called the Hadamard product), then added by the corresponding bias vector to obtain new self-attention outputs. Thus, we name the designed adapter as Hadamard adapter.

We carry out experiments on GLUE benchmark, including eight tasks. The experimental results demonstrate that the proposed Hadamard adapter achieves competitive performance with much fewer parameters than the existing fine-tuning methods. In addition, we take the learned parameter values of the Hadamard adapter as representations of downstream tasks. Through further analysis, we summarize some valuable tuning patterns for Hadamard adapter shared by various downstream tasks, which provide valuable guidance for further parameter reduction using shared adapters in future research.

To summarize, our contributions in this paper are threefold:

- Based on comprehensive empirical analysis, we design Hadamard adapter, which acts on self-attention outputs in PLMs with element-wise linear transformation. We also design an extreme parameter-efficient adapter tuning method based on the Hadamard adapter.
- We conduct extensive comparative experiments with several mainstream PLMs. The experimental results show that the proposed Hadamard adapter achieves the highest parametric efficiency in the fine-tuning history, and has competitive performance with full fine-tuning for various downstream tasks.
- We summarize some valuable tuning patterns for Hadamard adapter shared by various downstream tasks, which provide valuable guidance for further parameter reduction using shared adapters in future research.

2 EMPIRICAL ANALYSIS

To guide the design of our adapter tuning method, we conduct empirical studies that target at answering the three key questions as listed in the Introduction. In the following of this section, we first analyze the changes of self-attention outputs before and after full fine-tuning (for answering Q1), then we compare the difference among all fitting functions to select the suitable form of the Hadamard adapter (for answering Q2). Finally, we analyze the gradients of PLMs after fine-tuning on downstream tasks to select out the modules of great importance that would not be frozen in the adapter tuning (for answering Q3).

2.1 THE CHANGES OF SELF-ATTENTION OUTPUTS

We employ eight tasks in the GLUE benchmark to conduct the first analysis of how PLM's self-attention output changes before and after fine-tuning. For PLM, we adopt Roberta-large model, which has 24 hidden layers and outputs 1024-dimensional tensors in the

encoder, as an example to make analysis. Specifically, in order to compare the changes of self-attention outputs in each layer among all tasks, we adopt the norm of self-attention outputs instead of the original self-attention outputs. We analyze the distribution of the norm of self-attention outputs among all tasks before and after fine-tuning, and the changes during fine-tuning on each layer as shown in Fig. 1. The process is shown as the following equations:

$$\|\boldsymbol{A_b}\|_2 = \sqrt{\lambda_{max}(\boldsymbol{A_b}^T\boldsymbol{A_b})}, \quad \|\boldsymbol{A_a}\|_2 = \sqrt{\lambda_{max}(\boldsymbol{A_a}^T\boldsymbol{A_a})} \tag{1}$$

$$\Delta = \frac{\|\boldsymbol{A_a}\|_2 - \|\boldsymbol{A_b}\|_2}{\|\boldsymbol{A_b}\|_2} \tag{2}$$

where $\|\boldsymbol{A_b}\|_2$ and $\|\boldsymbol{A_b}\|_2$ represent the norm of self-attention outputs among all tasks before and after fine-tuning in a hidden layer, and $\lambda_{max}(\boldsymbol{A_b}^T\boldsymbol{A_b})$ is the eigenvalue of the matrix $\|\boldsymbol{A_b}\|_2$.

One box in Fig. 1(a)(b) and Fig. 1(c) represents the distribution of the norm of self-attention outputs and the corresponding changes in the layer, respectively. As can be observed in Fig. 1, the norm of self-attention outputs of all tasks significantly increase from an average of 60 to an average of 100 after fine-tuning, especially in the middle and back layer (Fig 1(a)(b)). After the fifteen layers, the changes become more significant as the number of layers increases, reaching the greatest changes at the last layer (Fig 1(c)). The above observations indicate that self-attention outputs change significantly during the fine-tuning process, which inspire us with the answer to Q1 as follows: *It is proper to inject an adapter right after the self-attention outputs to achieve similar performance gains with fine-tuning while updating much fewer parameters.*

2.2 FITTING FULL FINE-TUNING

We design fitting functions for self-attention outputs to make adapter tuning, which aims at letting the values of self-attention outputs approximate those in full fine-tuning of PLMs. We first optimize the parameters in the classifier modules. Next, we reload them and train different fitting functions, including linear function, quadratic function and higher order function (i.e. cubic function), respectively, to obtain new self-attention outputs. After that, we calculate the average value of each token in a sequence through dividing by the hidden size of the PLM in the new self-attention outputs as shown in Fig. 2(a). The process is shown in the following equations:

$$a'_j = \frac{1}{H}\sum_{i=1}^{H} a'_{ij} \tag{3}$$

where H represents the hidden size of a PLM. a'_{ij} is the value of the i^{th} dimension of the j^{th} token in the new self-attention outputs in a hidden layer based on a task. a'_j is the average value of each token in a sequence of a task. More detailed, we then calculate the average value of each sequence through dividing by sequence length in the new self-attention outputs. In this way, we obtain a characteristic value for each task which represents its respective average self-attention outputs. We analyze the distribution of the characteristic value among all tasks in each layer as shown in Fig. 2(b) with the process shown in the following equations:

$$a' = \frac{1}{L}\sum_{j=1}^{L} a'_j \tag{4}$$

where L represents the sequence length fed for the PLM. a' is the characteristic value of a task. We also analyze the average characteristic values of all tasks in each layer as shown in Fig. 2(c).

As shown in Fig. 2(a), dots of the same color represents the average value of each token in a sequence of all downstream tasks corresponding to one of the three fitting functions and fine-tuning. The dots of four settings are covered by each other, which also proves that fitting functions of different orders are similar in approximating the performance of fine-tuning. As shown in Fig. 2(b), one box represents the distribution of characteristic values among all downstream tasks in a layer. Median, quartile ranges which correspond to the characteristic value distribution of different fitting functions and fine-tuning are similar in each hidden layer. When we analyze the average characteristic value of all tasks, trends of linear function, quadratic function and higher order function are still similar, but slightly different from that of fine-tuning (Fig 2(c)). One dot represents the average characteristic values of all downstream tasks in a layer. As the order increases, the values between the fitting function and fine-tuning are closer, but the difference in distance can be ignored compared with the increase in the number of parameters. Therefore, we have answer to Q2 as follows: *A linear function is qualified enough to act on self-attention outputs to fit the performance of fine-tuning.*

2.3 GRADIENT ANALYSIS

We output the gradient and unit gradient of the top five layers in the first and last epoch during training, respectively, of a PLM (such as BERT-base model). Two representative datasets MRPC (similarity and paraphrase task, 3.7k) and SST-2 (single-sentence classification, 67k) from the GLUE benchmark are selected for analysis, and the results are shown in Table 1.

From the results of the gradients, we speculate that the classifier weights, embedding weights and intermediate weights of all tasks are more important in fine-tuning than the other modules because their gradients contribute the most in the first and last epochs. However, we find that some modules which contribute most during the fine-tuning have a large number of parameters, such as the intermediate module, which accounted for nearly half of all parameters. If we still fine-tune them, a lot of redundant computation is inevitable. We further analyze the results of unit gradient, which is the gradient divided by the number of parameters, and the results show that the classifier weights, embedded weights, and normalized weights are more important because their unit gradients contribute the most in the first and last epochs.

Therefore, we have answer to Q3 as follows: *We select out the classifier and the normalization as trainable modules in the adapter tuning.* We also make a theoretical analysis of the selection of these parameters as follows: i) The classifier makes prediction in the downstream tasks and directly affects the performance of tasks. ii) Normalization is to limit the data within a certain range and unify the distribution of each batch of training data. Since the input data distribution of each batch of the network is constantly changing, the changes of training pattern without normalization will make it difficult for the network to find a balance point, thus affecting the convergence of the network.

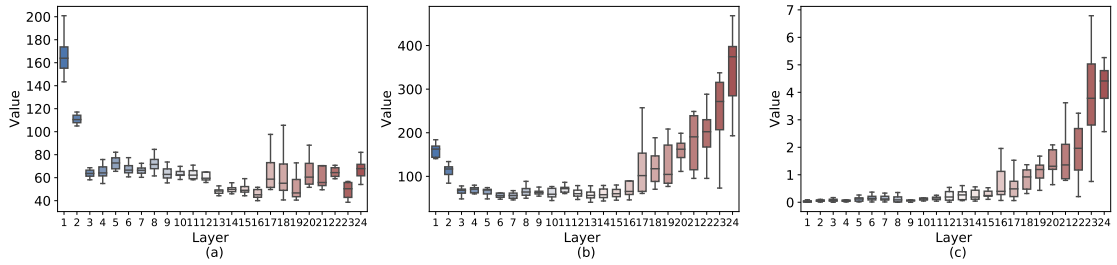

Figure 1: The distribution of the norm of the self-attention outputs among all tasks before (a) and after fine-tuning (b), and the corresponding changes (c) in each layer.

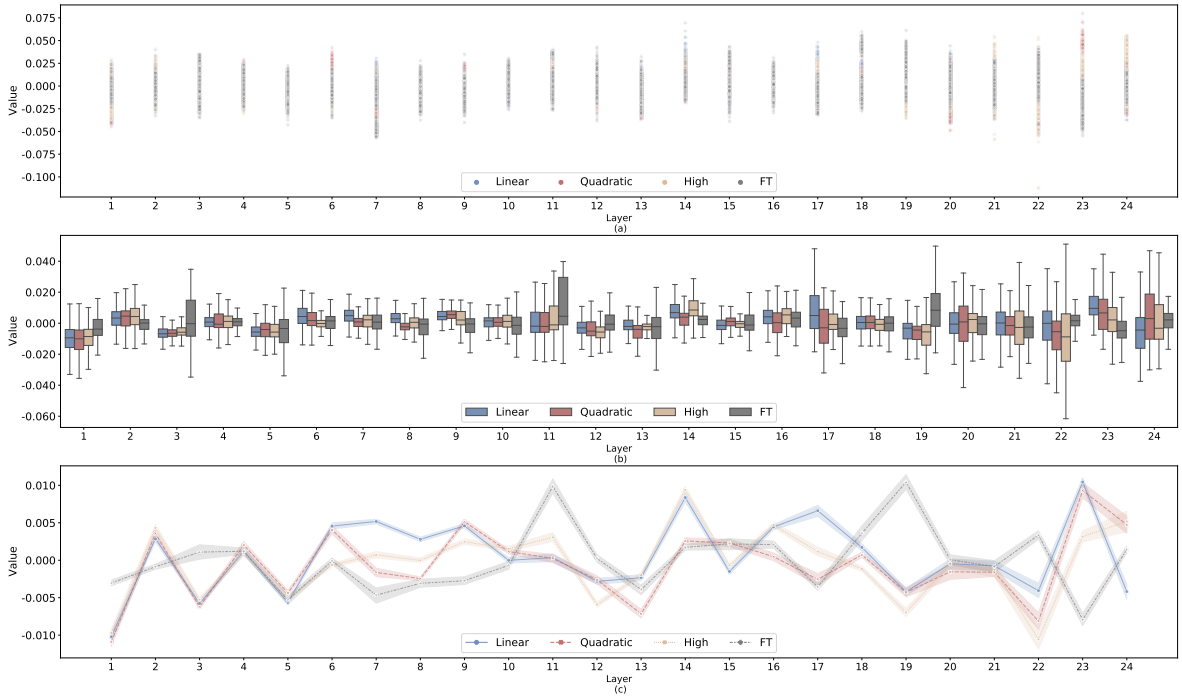

Figure 2: The average values of each token in a sequence of all tasks (a), the characteristic value distribution among all tasks (b), and the average characteristic values of all tasks (c) based on full fine-tuning and different fitting functions, respectively, in each hidden layer.

Table 1: The gradient and unit gradient of the top five layers (in descending order) in the first and last epoch, respectively. We adopt BERT-base model, MRPC and SST-2 dataset to show results and make analysis.

Task	Gradient in first epoch	Unit gradient in first epoch	Gradient in last epoch	Unit gradient in last epoch
MRPC	classifier.weight	classifier.bias	classifier.weight	classifier.bias
	embeddings.token_type_embeddings.weight	classifier.weight	embeddings.token_type_embeddings.weight	classifier.weight
	encoder.layer.4.intermediate.dense.weight	embeddings.token_type_embeddings.weight	pooler.dense.weight	embeddings.token_type_embeddings.weight
	encoder.layer.5.intermediate.dense.weight	encoder.layer.4.output.LayerNorm.bias	encoder.layer.4.intermediate.dense.weight	encoder.layer.4.output.LayerNorm.bias
	encoder.layer.4.attention.self.value.weight	encoder.layer.8.output.LayerNorm.weight	encoder.layer.7.intermediate.dense.weight	encoder.layer.3.output.LayerNorm.bias
SST-2	embeddings.token_type_embeddings.weight	classifier.bias	classifier.weight	classifier.bias
	embeddings.position_embeddings.weight	embeddings.token_type_embeddings.weight	embeddings.token_type_embeddings.weight	classifier.weight
	embeddings.word_embeddings.weight	classifier.weight	embeddings.position_embeddings.weight	embeddings.token_type_embeddings.weight
	encoder.layer.6.intermediate.dense.weight	embeddings.LayerNorm.bias	embeddings.word_embeddings.weight	encoder.layer.7.output.LayerNorm.weight
	encoder.layer.1.intermediate.dense.weight	encoder.layer.3.output.LayerNorm.weight	encoder.layer.8.intermediate.dense.weight	encoder.layer.7.output.LayerNorm.bias

3 METHODOLOGY

In this section, we first introduce the details of Hadamard adapter, and then clarify the process of the proposed parameter-efficient adapter tuning method in solving downstream tasks.

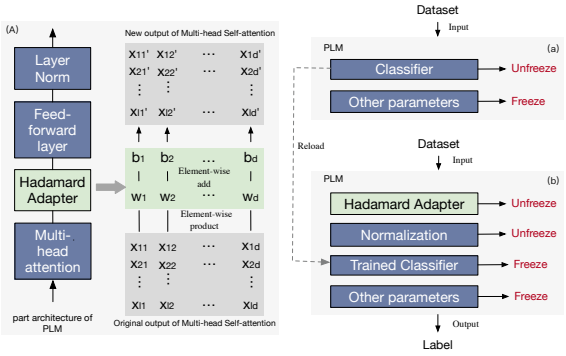

Figure 3: The framework of the Hadamard adapter (A), and the process of the parameter-efficient adapter tuning method, including two parts: (a) Train the classifier; (b) Inject the Hadamard adapter for self-attention outputs and unfreeze the normalization module.

3.1 THE HADAMARD ADAPTER

The framework of the Hadamard adapter is shown in Fig. 3(A). It is equivalent to a linear transformation based on feature dimension as shown in the following equation:

$$Adap: \boldsymbol{A'_{ij}} = \boldsymbol{W_j} * \boldsymbol{A_{ij}} + \boldsymbol{b_j} \tag{5}$$

Different feature dimensions correspond to different linear transformation parameters, but different positions (i.e. each token in a sequence) share the same parameters. We put such an adapter in each layer of the PLMs. Adapters of different layers have different parameters. Based on the empirical analysis in Sec. 2.1, we put this adapter right after the self-attention outputs. Specifically, we multiply the weight vector $\boldsymbol{W_j}$ of the Hadamard adapter to the self-attention outputs $\boldsymbol{A_{ij}}$, and then add the bias vector $\boldsymbol{b_j}$ to the product to obtain new self-attention outputs $\boldsymbol{A'_{ij}}$. The weight vector and the bias vector in the Hadamard adapter are both 1-d vectors of the same shape as the hidden size of the PLM, such as 768 for the base version or 1024 for the large version for BERT model. The number of the Hadamard adapter in a PLM is also the same as the number of PLMs' layers, such as 12 for the base version and 24 for the large version in BERT model. All weight vectors are initialized as 1.0 and all bias vectors are initialized as 0.0. The initial value is equivalent to not adding any adapter. The approximate number of parameters of this adapter is from 30,000 to 100,000 according to different size of PLMs and 3,000 to 4,000 in each layer of PLMs, which is only 0.03% of the full fine-tuning.

3.2 THE ADAPTER TUNING METHOD

The parameter-efficient adapter tuning method is to inject adapters to PLMs and make continuous fine-tuning as shown in Fig. 3(a)(b). Specifically, we first only unfreeze and train the pooling and classifier modules on downstream tasks. Next, we inject the Hadamard adapter right after self-attention outputs. After that, we reload the trained pooling and classifier layers, and only fine-tune the Hadamard adapter and the normalization module in each layer. Although the above-mentioned two-stage training process can be time-consuming and computationally expensive, the performance is better than joint training in our experiments. We have analyze the possible reason in the paper that the proposed Hadamard Adapter has not been optimized in the pre-training stage which will affect the effect of classifier layer. We also find other research, such as LP-FT [17], demonstrating the better performance of two-stage training.

Train the classifier module. For each given downstream task, we first learn only the classifier module on the training dataset, including the pooling and linear output layers. While learning the classifier module, the other parameters of the PLMs are frozen. Since all layers except the classifier module do not participate in backward propagation and gradient update, they can be shared by different tasks. In addition, the computational cost of this step is very small compared to the overall cost of full fine-tuning.

Inject the Hadamard adapter and unfreeze the normalization module. Secondly, we inject the Hadamard adapter right after the self-attention outputs. Multi-head self-attention is an important mechanism in PLMs, which first obtains representation of queries Q, keys K and values V, and then makes scaled dot-product for $heads = 16$ times to obtain self-attention score of each attention head. Outputs of all attention heads are finally concatenated to obtain self-attention output A_i. Based on multi-head self-attention, we first transform the 3-D dimensional self-attention outputs into 2-D dimensional matrices, that is, making the first and second dimension flatten. Next, we input the 2-D dimensional self-attention outputs into the Hadamard adapter and transform the self-attention outputs into 3-D dimensional matrix A'_i. The process is as follows:

$$Q_l^i = Q\boldsymbol{W_l^Q}, K_t^i = K\boldsymbol{W_l^K}, V_t^i = V\boldsymbol{W_l^V}, \quad A_l^i = softmax(\frac{Q_l^i {K_l^i}^T}{\sqrt{d_k}})V_l^i \tag{6}$$

$$A_i = Concat(A_1^i, \cdots, A_T^i), \quad A'_i = Adap(A_i) \tag{7}$$

After that, We reload the parameters of the trained classifier module and only unfreeze the normalization module besides the Hadamard adapter. Based on the empirical analysis in Sec. 2.3, we argue that the normalization module is very important to improve the accuracy of downstream tasks. The reason is that the value range of the self-attention outputs changes after being transformed with the Hadamard adapter. Obviously, it is necessary to relearn the parameters of the normalized module and make them adapt to the value distribution of the new self-attention outputs in order to achieve better results. In order to further scale the parameter size and reach the limit of parameter efficiency, we only unfreeze the normalization module right after the intermediate outputs rather than unfreeze that right after the self-attention output.

4 EXPERIMENT

This section reports the experiments conducted on GLUE benchmark in validating the effectiveness of the proposed Hadamard adapter on different SOTA PLMs, as well as comparing the performance with other parameter-efficient methods.

4.1 EXPERIMENTAL SETUP

The experiments are carried out on Tesla V100 GPUs with Pytorch in Python. Similar with previous work [6, 8, 13, 19, 26], the batch

size is set to 16 or 32, and the sequence length is set to 128. We maintain the hyper parameters such as 0.01 for weight decay, 0.9 for β_1, and 0.999 for β_2, etc. We set the learning rate from 2e-03 to 4e-03 in training the classifier module, from 2e-05 to 4e-05 in full fine-tuning, and from 1e-03 to 9e-03 in training with the Hadamard adapter to obtain the best performance in each training setting. All tasks are training for 20 epochs with each PLM.

Datasets and Baselines. We adopt GLUE benchmark [38], including eight tasks from different domains, as the datasets. They are divided into single-sentence classification (CoLA, SST-2), similarity and paraphrase (MRPC, STS-B, QQP), and inference (MNLI, QNLI, RTE). We ignore the WNLI dataset whose data amount is too small. Same as the previous research [3, 16] on GLUE benchmark, we also adopt Matthews correlation coefficient [1] and Pearson coefficient [2] to evaluate the performance on the CoLA dataset and STS-B dataset, respectively, and use accuracy for other datasets. We carry out our experiments on several SOTA PLMs, including BERT [8], RoBERTa [26], BART [19], DeBERTa [13], and ELECTRA [6]. Moreover, we use powerful parameter-efficient adapters shown in Table 3 as baselines to compare the parameter amount and performance on GLUE benchmark.

4.2 MAIN RESULTS

We first report the performance of the classifier, Hadamard adapter and full fine-tuning in GLUE benchmark based on several SOTA PLMs in Table 2, where the results all come from our own machines, which represent only fine-tuning the classifier module, the proposed adapter tuning method, and full fine-tuning, respectively. We find that only training the classifier module achieves 77.5% in performance compared with that of the corresponding full fine-tuning averagely in selected PLMs and tasks. It represents that the classifier module is important for solving downstream tasks, so it is necessary to be fine-tuned alone. Next, we reload the trained classifier module, inject the Hadamard adapter for self-attention outputs, and only unfreeze the normalization module. The performance increase a large degree, which achieves 99.4% compared with that of full fine-tuning averagely in selected PLMs and tasks. Some results are even better than the corresponding full fine-tuning, such as the MPRC dataset with the BERT-base model. It represent that the Hadamard adapter has a stunning effect which has competitive performance and achieve extreme parameter efficiency (0.033% parameters of full fine-tuning).

We also compare the performance between the proposed adapter tuning method with Hadamard adapter and other parameter-efficient baselines as shown in Table 3, where baselines' results are replicated from their corresponding published papers. As can be observed, there is not much difference in performance, but the Hadamard adapter uses the fewest amount of parameters. This comparison results prove that the Hadamard adapter could uses the fewest parameters to achieve competitive performance with full fine-tuning and the existing parameter-efficient tuning methods.

[1] https://en.wikipedia.org/wiki/Phi_coefficient
[2] https://en.wikipedia.org/wiki/Pearson_correlation_coefficient

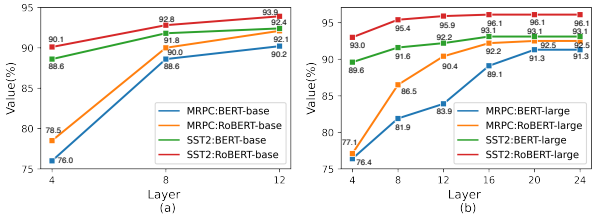

Figure 4: The influence of different number of unfreezing layers of the Hadamard adapter on the performance of the Hadamard adapter with model of base version (a) and large version (b).

4.3 ABLATION STUDY

In this part, we carry out ablation study to clarify the effect of each module in the Hadamard adapter as shown in Table 4, as well as the influence of different number of unfreezing layers to the performance as shown in Fig. 4.

The effect of each module. The results for the effect of each module are shown in Table 4. We first unfreeze any one module of the adapter tuning method (see row 2-5). The results represent that the bias vectors in the Hadamard adapter and the normalization module contribute more than the weight vectors (see row 2-5). We also unfreeze the normalization module (those right after intermediate outputs, short as normalization module) and attention-based normalization module (those right after self-attention outputs, short as attention-based normalization module), respectively. We find that the coarse-grained normalization modules (i.e. normalization module) are more necessary than the fine-grained normalization modules (i.e. attention-based normalization module) (see row 4-5). Next, we unfreeze any two modules of the adapter tuning method (see row 6-10). The results represent that the most two effective modules, i.e., bias vectors in the Hadamard adapter and the normalization module, also achieve the best performance after being unfrozen simultaneously. After that, we unfreeze three or four modules of the adapter tuning method (see row 11-13). Compared with the final three modules, we observe that when we add the attention-based normalization module, the performance decreases a little bit. It represents that some parameters are not valuable enough to improve PLMs' performance in the downstream tasks.

The effect of the number of unfreezing layers. The results for the number of unfreezing layers of the Hadamard adapter on downstream tasks are shown in Table 5. We select two tasks to visualize the trend as shown in Fig. 4. We find that as we unfreeze more layers, the performance increase consistently. And they achieve satisfying performance when we unfreeze over a half of all layers (8 for models of base version and 16 for those of large version). The experiment inspires us that parameters in some layers of the Hadamard adapter are still redundant which can be removed to achieve more parameter-efficient with 0.022% paramters, but this conjecture need to be validated with more datasets and PLMs.

5 EXPLORATORY ANALYSIS

To get deeper understanding on the tuning results and provide guidance for better applying Hadamard adapter to various downstream

Table 2: Performance of training classifier module, adapter tuning and full fine-tuning in GLUE benchmark based on several SOTA PLMs.

PLMs	Training type	MRPC	CoLA	MNLI	QNLI	QQP	RTE	SST-2	STS-B	Average
BERT-base	Classifier	71.8	37.0	54.4	70.6	79.3	57.4	87.4	60.3	64.8
	Hadamard adapter	**90.2**	**58.4**	**80.4**	**89.7**	**85.9**	**71.9**	**92.4**	**88.5**	**82.2**
	Full fine-tuning	89.4	56.5	83.9	91.3	87.5	64.6	93.0	88.6	81.9
BERT-large	Classifier	72.2	37.9	61.7	71.8	79.8	58.1	89.7	61.9	66.6
	Hadamard adapter	**91.3**	**62.7**	**83.6**	**91.1**	**87.2**	**73.1**	**93.1**	**90.0**	**84.0**
	Full fine-tuning	90.2	62.8	85.6	92.1	88.5	71.9	93.1	90.1	84.3
RoBERTa-base	Classifier	73.1	44.4	59.3	70.9	75.4	58.5	83.6	61.9	65.9
	Hadamard adapter	**92.1**	**64.3**	**85.5**	**91.6**	**87.7**	**80.7**	**93.9**	**90.1**	**85.7**
	Full fine-tuning	91.9	64.4	86.7	92.8	89.0	78.0	94.7	91.1	86.1
RoBERTa-large	Classifier	72.8	45.9	62.0	70.6	77.3	58.8	88.4	62.4	67.3
	Hadamard adapter	**92.5**	**67.8**	**89.5**	**95.1**	**90.3**	**85.9**	**96.1**	**91.9**	**88.6**
	Full fine-tuning	92.7	65.9	91.0	94.5	89.7	86.3	96.2	91.7	88.5
BART-base	Classifier	71.5	43.7	60.1	70.0	76.1	56.9	87.3	61.2	65.9
	Hadamard adapter	**86.2**	**61.7**	**85.5**	**92.8**	**86.8**	**76.5**	**94.1**	**89.4**	**84.1**
	Full fine-tuning	87.1	62.0	87.8	93.9	88.3	77.7	95.0	91.2	85.4
BART-large	Classifier	72.3	44.3	61.8	71.4	78.0	58.4	88.9	61.8	67.1
	Hadamard adapter	**88.1**	**65.0**	**87.4**	**94.0**	**89.0**	**82.3**	**95.7**	**91.3**	**86.6**
	Full fine-tuning	88.2	63.4	88.0	95.5	90.1	79.3	95.9	91.9	86.5
DeBERTa-base	Classifier	72.2	45.2	62.1	70.3	76.6	58.8	87.8	63.5	67.1
	Hadamard adapter	**89.0**	**65.8**	**90.1**	**94.7**	**89.2**	**85.4**	**94.4**	**92.4**	**87.6**
	Full fine-tuning	91.3	66.2	91.2	96.1	90.2	87.4	96.1	93.0	88.9
DeBERTa-large	Classifier	73.5	46.3	63.9	72.3	77.9	60.2	89.1	64.1	68.4
	Hadamard adapter	**90.4**	**67.3**	**92.4**	**95.9**	**89.8**	**86.7**	**96.8**	**93.0**	**89.0**
	Full fine-tuning	92.5	68.0	93.3	96.9	90.5	88.3	97.1	93.5	90.0
ELECTRA-base	Classifier	72.9	47.3	63.3	71.3	76.6	60.2	89.5	65.0	68.3
	Hadamard adapter	**89.4**	**68.2**	**91.2**	**96.1**	**87.7**	**85.6**	**96.2**	**93.5**	**88.5**
	Full fine-tuning	90.1	69.0	92.9	96.4	88.5	86.3	96.9	93.7	89.2
ELECTRA-large	Classifier	74.0	47.1	64.8	72.9	78.3	61.2	90.3	66.8	69.4
	Hadamard adapter	**92.0**	**68.5**	**92.9**	**96.1**	**89.9**	**87.2**	**97.1**	**94.2**	**89.7**
	Full fine-tuning	93.0	69.3	93.9	96.4	91.4	88.7	97.1	94.9	90.6

Table 3: Comparison between tuning with Hadamard adapter and other adapters in GLUE benchmark in several SOTA PLMs.

PLMs	Adapter	Parameters	MRPC	CoLA	MNLI	QNLI	QQP	RTE	SST-2	STS-B	Average
BERT-base	Hadamard adapter	**0.03%**(↓ 0.06%)	90.2	58.4	80.4	89.7	**85.9**	71.9	**92.4**	88.5	82.2(↓ 0.2)
	BitFit [3]	0.09%	**90.4**	**58.8**	**81.8**	**90.2**	84.0	**72.3**	92.1	**89.2**	**82.4**
BERT-large	Hadamard adapter	**0.03%**(↓ 0.05%)	91.3	62.7	83.6	91.1	**87.2**	73.1	93.1	90.0	84.0(↓ 0.2)
	BitFit	0.08%	**91.7**	**63.6**	84.6	**91.4**	85.4	**73.2**	93.2	**90.3**	**84.2**
	Adapters (8-256) [15]	14.44%	89.5	59.5	**85.0**	90.7	71.8	71.5	94.0	86.9	80.0
	Adapters (64) [15]	13.33%	89.6	56.9	**85.0**	**91.4**	71.8	68.8	**94.2**	87.3	79.6
RoBERTa-base	Hadamard adapter	**0.03%**(↓ 0.21%)	92.1	**64.3**	85.5	91.6	87.7	80.7	93.9	90.1	85.7(↓ 1.5)
	BitFit	0.08%	**92.7**	62.0	84.7	91.8	84.0	81.5	93.7	90.8	85.2
	Adpt^D [32]	0.24%	88.5	60.8	87.1	93.1	90.2	71.5	94.2	89.7	84.4
	Adpt^D	0.72%	88.4	62.6	87.3	93.0	90.6	75.9	94.7	90.3	85.4
	LoRA [16]	0.24%	89.7	63.4	**87.5**	**93.3**	**90.8**	**86.6**	**95.1**	**91.5**	**87.2**
RoBERTa-large	Hadamard adapter	**0.03%**(↓ 0.2%)	92.5	67.8	89.5	**95.1**	90.3	85.9	96.1	91.9	88.6(-)
	Adpt^P [29]	0.85%	90.2	**68.3**	90.2	94.8	91.9	83.8	96.1	92.1	88.4
	Adpt^P	0.23%	89.7	67.8	90.5	94.8	91.7	80.1	**96.6**	91.9	87.9
	Adpt^H [15]	1.70%	88.7	66.5	89.9	94.7	**92.1**	83.4	89.9	91.0	87.8
	Adpt^H	0.23%	87.7	66.3	90.3	94.7	91.5	72.9	90.3	91.5	86.4
	LoRA	0.23%	90.2	68.2	**90.6**	94.8	91.6	85.2	90.6	**92.3**	**88.6**
	RoBERTa-AT [25]	0.85%	**92.9**	67.4	90.4	94.7	88.5	83.4	96.3	-	87.7
	RoBERTa-WARP [25]	0.28%	91.2	60.6	88.2	93.5	84.5	**86.3**	96.0	-	85.8
	RoBERTa-YT [25]	4.60%	85.0	54.4	83.1	88.2	87.4	81.9	94.5	-	82.1
BART-large	Hadamard adapter	**0.02%**(↓ 7.71%)	**88.1**	**65.0**	**87.4**	**94.0**	**89.0**	**82.3**	**95.7**	-	86.6(↑ 9.7)
	BARTen-FbT [25]	8.52%	76.0	42.1	81.9	88.4	86.7	60.6	93.2	-	75.6
	BARTen-YT [25]	7.73%	79.2	44.4	82.3	88.2	85.5	62.8	94.4	-	**76.9**

tasks, we would like to visualize and analyze each module in the Hadamard adapter of each layer among different downstream tasks as shown in Fig 5.

The overall analysis aims to answer the following three questions: i) For various downstream tasks, which layer of the adapter has the greater weight and bias variation? (Note that small variation presents the consistency among the learned adapters on different

Table 4: The effect of each module in the Hadamard adapter on other downstream tasks based on BERT-base model. W: Weight, B: Bias, N: Norm, A: Att-Norm.

Module	MRPC	SST-2	CoLA	QNLI	QQP	MNLI	RTE	STS-B
W	73.1	88.7	55.0	84.8	82.4	77.6	67.3	84.7
B	81.0	91.1	56.9	88.0	84.9	79.6	68.7	86.0
N	80.2	90.8	56.6	87.5	84.4	79.3	68.4	85.7
A	79.6	90.5	56.5	87.3	84.1	79.0	68.2	85.5
W+A	80.9	91.0	56.8	88.1	84.7	79.5	68.8	86.2
W+N	81.9	91.3	57.2	88.3	85.0	79.6	69.0	86.3
B+A	81.7	91.4	57.1	88.4	85.0	79.7	69.1	86.5
B+N	82.1	91.7	57.2	88.6	85.2	79.8	69.3	86.7
W+B	78.2	90.0	56.0	86.8	83.6	78.5	67.8	85.0
W+B+N+A	82.6	92.0	57.3	88.9	85.4	80.0	69.8	87.2
W+B+A	82.8	92.1	57.8	89.1	85.2	79.8	70.0	87.9
(Ours)	**83.7**	**92.4**	**58.4**	**89.7**	**85.9**	**80.4**	**71.9**	**88.5**

downstream tasks.) ii) What is the difference between normalized module distribution under adapter tuning and full fine-tuning for various downstream tasks? and iii) What are the commonalities of weight and bias in adapter between different downstream tasks? Here we take RoBERTa-large model as an example to conduct the analysis, and the other PLMs are observed to have similar conclusions.

For the first question, we find that the weight and bias vectors of all datasets basically vary around 1.0 and 0.0 in each layer, respectively (see Fig 5 (a_1)(a_2)). One box represents the distribution of the weight and bias vector values among all downstream tasks in the corresponding layer, respectively. in two sub-figures. From the box plot, we find that the variance and extremum (maximum or minimum values) of weight and bias vectors in different layers are similar, respectively. The results represent that *the consistency degree of the learned Hadamard adapters among different tasks is similar in different layers.*

For the second question, the weight vectors of the subsequent normalization module both vary around 1.0 after adapter tuning (see Fig 5 (b_1), one box represents the distribution of the weight vector values among all downstream tasks in a layer after adapter tuning) and full fine-tuning (see Fig 5 (b_2), one box represents the counterparts after full fine-tuning). In the adapter tuning, we also find that the variance and extremum of weight vectors in different layers are similar Different from them, in the full fine-tuning, both the most volatile layers and the most changes in extremum are the front layers, and the corresponding fewest ones are both the back layers. Moreover, the bias vectors of the normalization module both vary around 0.0 after adapter tuning (see Fig 5 (b_3), one box represents the distribution of the bias vector values among all downstream tasks in a layer after adapter tuning) and full fine-tuning (see Fig 5 (b_4), one box represents the counterparts after full fine-tuning). We find that the trend of most volatile and extremum in adapter tuning and full fine-tuning are similar. The most volatile layers and the most changes in extremum are both the front layers, and the corresponding fewest ones are both the back layers. The results represent that *the trend of the consistency degree among different tasks along with layers in full fine-tuning and adapter tuning is general similar.*

For the third question, we calculate the cosine similarity of weight vectors (see Fig 5 (c_1)) and bias vectors (see Fig 5 (c_2)) in the Hadamard adapter, respectively, between each two downstream tasks. Red color represents more similar between weight or bias vectors of two tasks and blue color represents less similar. Due to limitation of space, we only display the heatmaps of the first, the middle layer and the average results. From the heatmaps, we find that the similarity of weight vectors in each layer for different tasks are almost same and consistent, and the values are closed to 1.0. Meanwhile, the similarity of bias vectors in each layer are obviously different, and achieve 0.3 at most. The results indicate that bias contributes most in the Hadamard adapter and some adapter weights can be reused across different tasks, potentially leading to a more efficient and generalizable adapter tuning approach. The implications of this finding are significant, as it suggests that using a shared adapter approach could provide a more efficient and effective way to fine-tune pre-trained models for multiple tasks. By sharing weight vectors across tasks, the adapter network can be made smaller and less complex, reducing the risk of overfitting and improving the model's generalization performance. Further research is required to explore the extent to which adapters can be shared across tasks.

6 RELATED WORK

Previous parameter-efficient fine-tuning of PLMs contains three main categories of methods, i.e., adapter tuning, prefix tuning, and prompt tuning. Adapter tuning [15] is to inject a small neural network module in each layer of PLMs, and only fine-tune parameters in this small neural network module. For instance, Hu et al. [16] propose Low-Rank Adaptation (LoRA) which injects trainable rank decomposition matrices into each layer of the Transformer architecture. Mahabadi et al. [27] learn adapter parameters for all layers and tasks by generating them using shared hypernetworks in a transformer model. Liu et al. [22] introduce IA3, which incorporates three vectors (lk, lv, and lff) in each Transformer layer block, going beyond the simple addition of weight and bias vectors to the self-attention outputs. Qi et al. [30] propose LN-tuning, which focuses on keeping only the gain term and bias term trainable in the LayerNorm module. Prefix tuning [20] and prompt tuning [18] preset additional adjustable prefix tokens in the input layers or hidden layers, and only train these soft prompts during the fine-tuning on downstream tasks. For instance, Liu et al. [24] propose a prefix tuning method P-tuning which improve GPTs and BERTs performance in both few-shot and fully supervised settings. Sun et al. [34] propose the black-box tuning framework to optimize the continuous prompt prepended to the input text via derivative-free optimization. Gu et al. [10] pre-train prompts by adding soft prompts into the pre-training stage to obtain a better initialization. He et al. [12] propose a MAM Adapter which use prefix tuning with a small bottleneck dimension at the attention sub-layers.

In addition to the above three parameter-efficient fine-tuning ways, there are also some other related research. For example, Ben Zaken et al. [3] propose a Bias-term Fine-Tuning method which train only the bias-terms and the task-specific classification layer. Liu et al. [23] present a novel PLMs' compression approach based on the matrix product operator. Ansell et al. [1] propose Lottery Ticket Sparse Fine-Tuning conceived for pruning of large neural networks. Liu et al. [25] learns dense representations for labels Y

Table 5: The influence of different number of unfreezing layers of the Hadamard adapter on the performance of downstream tasks.

PLMs	Tasks	4	8	12	16	20	24	PLMs	Tasks	4	8	12	16	20	24
BERT-base	CoLA	50.0	56.3	**58.4**	-	-	-	RoBERTa-base	CoLA	55.7	62.6	**64.3**	-	-	-
	QNLI	80.8	87.5	**89.7**	-	-	-		QNLI	83.3	89.2	**91.6**	-	-	-
	QQP	78.2	84.0	**85.9**	-	-	-		QQP	79.6	85.9	**87.7**	-	-	-
	MNLI	71.5	78.3	**80.4**	-	-	-		MNLI	77.5	83.7	**85.5**	-	-	-
	RTE	62.5	69.8	**71.9**	-	-	-		RTE	70.3	78.8	**80.7**	-	-	-
	STS-B	79.0	86.7	**88.5**	-	-	-		STS-B	81.2	87.9	**90.1**	-	-	-
BERT-large	CoLA	53.4	57.9	61.8	62.0	62.7	**62.7**	RoBERTa-large	CoLA	58.1	59.7	62.0	65.8	66.9	**67.8**
	QNLI	83.4	84.7	88.5	91.1	91.1	**91.1**		QNLI	85.2	90.6	93.8	95.1	95.1	**95.1**
	QQP	80.6	82.4	85.5	86.9	87.2	**87.2**		QQP	82.4	86.5	88.7	90.3	90.3	**90.3**
	MNLI	73.7	80.5	82.3	82.9	82.9	**83.6**		MNLI	79.6	83.1	85.4	87.2	89.5	**89.5**
	RTE	65.8	67.3	70.8	73.1	73.1	**73.1**		RTE	72.1	75.6	80.5	83.0	84.7	**85.9**
	STS-B	82.3	83.2	86.4	89.5	89.8	**90.0**		STS-B	82.8	84.3	88.4	90.9	91.9	**91.9**

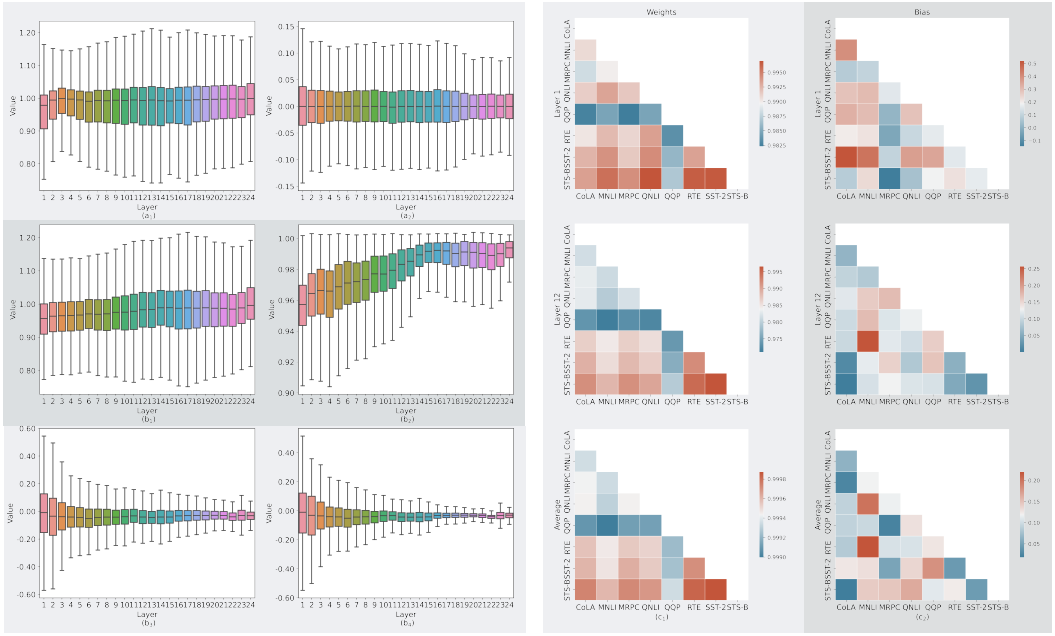

Figure 5: Each module in the Hadamard adapter based on each layer to answer three questions. Question one corresponds to (a_1) and (a_2), Question 1 corresponds to from (b_1) to (b_4), and Question 3 corresponds to (c_1) and (c_2)

when training PLMs and aligns them to fixed feature representation. Xu et al. [40] only update a subset of parameters of PLMs via masking out the gradients of the non-child network during the backward process. Mao et al. [28] introduce UNIPELT that learns to activate (upweight) the submodules that best suit the current task or specific data sample and deactivate (downweight) the rest.

While the existing research significantly enhances the parameter efficiency when training PLMs, we hypothesize that there still exist superfluous parameters that could be eliminated. To address this, we introduce a highly efficient Hadamard adapter, which boasts the fewest parameters to date, yet delivers performance on par with full fine-tuning methods.

7 CONCLUSIONS AND FUTURE WORK

In this paper, we propose a Hadamard adapter based on a simple but effective element-wise linear transformation on the outputs of self-attention in PLMs. We make comprehensive analysis for the feasibility of the Hadamard adapter, and summarize out some valuable patterns of it on downstream tasks. The experiments demonstrate that the proposed Hadamard adapter achieves competitive performance with 0.033% parameters compared with full fine-tuning. Moreover, some layers in the Hadamard adapter are considered redundant to be removed for more parameter efficiency with 0.022% parameters, which will be further investigated in the future study.

8 ACKNOWLEDGEMENT

Some of computational resource are partially supported by Shanghai Municipal Science and Technology Major Project (No.2021SHZDZX0103), Science and Technology Commission of Shanghai Municipality Grant (No. 22511105902), National Key Research and Development Project (No.2020AAA0109302), National Natural Science Foundation of China (No.62072323), Shanghai Science and Technology Innovation Action Plan (No. 22511104700, 22511105902), Shanghai Municipal Science and Technology Major Project (No.2021SHZDZX0103), and Science and Technology Commission of Shanghai Municipality Grant (No. 22511105902).

REFERENCES

[1] Alan Ansell, Edoardo Maria Ponti, Anna Korhonen, and Ivan Vulić. 2021. Composable Sparse Fine-Tuning for Cross-Lingual Transfer. https://doi.org/10.48550/ARXIV.2110.07560

[2] Lei Jimmy Ba and Rich Caruana. 2013. Do Deep Nets Really Need to be Deep? https://doi.org/10.48550/ARXIV.1312.6184

[3] Elad Ben Zaken, Yoav Goldberg, and Shauli Ravfogel. 2022. BitFit: Simple Parameter-efficient Fine-tuning for Transformer-based Masked Language-models. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*. Association for Computational Linguistics, Dublin, Ireland, 1–9. https://doi.org/10.18653/v1/2022.acl-short.1

[4] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared 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 M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models are Few-Shot Learners. https://doi.org/10.48550/ARXIV.2005.14165

[5] Yu Cheng, Duo Wang, Pan Zhou, and Tao Zhang. 2017. A Survey of Model Compression and Acceleration for Deep Neural Networks. https://doi.org/10.48550/ARXIV.1710.09282

[6] Kevin Clark, Minh-Thang Luong, Quoc V. Le, and Christopher D. Manning. 2020. ELECTRA: Pre-training Text Encoders as Discriminators Rather Than Generators. https://doi.org/10.48550/ARXIV.2003.10555

[7] Jeffrey Dean, G.s Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Quoc Le, Mark Mao, Aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, and Andrew Ng. 2012. Large Scale Distributed Deep Networks. *Advances in neural information processing systems* (10 2012).

[8] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. https://doi.org/10.48550/ARXIV.1810.04805

[9] Yunchao Gong, Liu Liu, Ming Yang, and Lubomir Bourdev. 2014. Compressing Deep Convolutional Networks using Vector Quantization. https://doi.org/10.48550/ARXIV.1412.6115

[10] Yuxian Gu, Xu Han, Zhiyuan Liu, and Minlie Huang. 2021. PPT: Pre-trained Prompt Tuning for Few-shot Learning. https://doi.org/10.48550/ARXIV.2109.04332

[11] Song Han, Jeff Pool, John Tran, and William J. Dally. 2015. Learning both Weights and Connections for Efficient Neural Networks. https://doi.org/10.48550/ARXIV.1506.02626

[12] Junxian He, Chunting Zhou, Xuezhe Ma, Taylor Berg-Kirkpatrick, and Graham Neubig. 2021. Towards a unified view of parameter-efficient transfer learning. *arXiv preprint arXiv:2110.04366* (2021).

[13] Pengcheng He, Xiaodong Liu, Jianfeng Gao, and Weizhu Chen. 2020. DeBERTa: Decoding-enhanced BERT with Disentangled Attention. https://doi.org/10.48550/ARXIV.2006.03654

[14] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. 2015. Distilling the Knowledge in a Neural Network. https://doi.org/10.48550/ARXIV.1503.02531

[15] Neil Houlsby, Andrei Giurgiu, Stanislaw Jastrzebski, Bruna Morrone, Quentin de Laroussilhe, Andrea Gesmundo, Mona Attariyan, and Sylvain Gelly. 2019. Parameter-Efficient Transfer Learning for NLP. In *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA (Proceedings of Machine Learning Research, Vol. 97)*, Kamalika Chaudhuri and Ruslan Salakhutdinov (Eds.). PMLR, 2790–2799. http://proceedings.mlr.press/v97/houlsby19a.html

[16] Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2021. LoRA: Low-Rank Adaptation of Large Language Models. https://doi.org/10.48550/ARXIV.2106.09685

[17] Ananya Kumar, Aditi Raghunathan, Robbie Jones, Tengyu Ma, and Percy Liang. 2022. Fine-tuning can distort pretrained features and underperform out-of-distribution. *arXiv preprint arXiv:2202.10054* (2022).

[18] Brian Lester, Rami Al-Rfou, and Noah Constant. 2021. The Power of Scale for Parameter-Efficient Prompt Tuning. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, Online and Punta Cana, Dominican Republic, 3045–3059. https://doi.org/10.18653/v1/2021.emnlp-main.243

[19] Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Ves Stoyanov, and Luke Zettlemoyer. 2019. BART: Denoising Sequence-to-Sequence Pre-training for Natural Language Generation, Translation, and Comprehension. https://doi.org/10.48550/ARXIV.1910.13461

[20] Xiang Lisa Li and Percy Liang. 2021. Prefix-Tuning: Optimizing Continuous Prompts for Generation. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*. Association for Computational Linguistics, Online, 4582–4597. https://doi.org/10.18653/v1/2021.acl-long.353

[21] Zhaojiang Lin, Andrea Madotto, and Pascale Fung. 2020. Exploring Versatile Generative Language Model Via Parameter-Efficient Transfer Learning. In *Findings of the Association for Computational Linguistics: EMNLP 2020*. Association for Computational Linguistics, Online, 441–459. https://doi.org/10.18653/v1/2020.findings-emnlp.41

[22] Haokun Liu, Derek Tam, Mohammed Muqeeth, Jay Mohta, Tenghao Huang, Mohit Bansal, and Colin A Raffel. 2022. Few-shot parameter-efficient fine-tuning is better and cheaper than in-context learning. *Advances in Neural Information Processing Systems* 35 (2022), 1950–1965.

[23] Peiyu Liu, Ze-Feng Gao, Wayne Xin Zhao, Zhi-Yuan Xie, Zhong-Yi Lu, and Ji-Rong Wen. 2021. Enabling Lightweight Fine-tuning for Pre-trained Language Model Compression based on Matrix Product Operators. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*. Association for Computational Linguistics, Online, 5388–5398. https://doi.org/10.18653/v1/2021.acl-long.418

[24] Xiao Liu, Yanan Zheng, Zhengxiao Du, Ming Ding, Yujie Qian, Zhilin Yang, and Jie Tang. 2021. GPT Understands, Too. https://doi.org/10.48550/ARXIV.2103.10385

[25] Yitao Liu, Chenxin An, and Xipeng Qiu. 2022. $\mathcal{Y}$-Tuning: An Efficient Tuning Paradigm for Large-Scale Pre-Trained Models via Label Representation Learning. https://doi.org/10.48550/ARXIV.2202.09817

[26] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. RoBERTa: A Robustly Optimized BERT Pretraining Approach. https://doi.org/10.48550/ARXIV.1907.11692

[27] Rabeeh Karimi Mahabadi, Sebastian Ruder, Mostafa Dehghani, and James Henderson. 2021. Parameter-efficient Multi-task Fine-tuning for Transformers via Shared Hypernetworks. https://doi.org/10.48550/ARXIV.2106.04489

[28] Yuning Mao, Lambert Mathias, Rui Hou, Amjad Almahairi, Hao Ma, Jiawei Han, Wen-tau Yih, and Madian Khabsa. 2021. Unipelt: A unified framework for parameter-efficient language model tuning. *arXiv preprint arXiv:2110.07577* (2021).

[29] Jonas Pfeiffer, Aishwarya Kamath, Andreas Rücklé, Kyunghyun Cho, and Iryna Gurevych. 2020. AdapterFusion: Non-Destructive Task Composition for Transfer Learning. (2020). https://doi.org/10.48550/ARXIV.2005.00247

[30] Wang Qi, Yu-Ping Ruan, Yuan Zuo, and Taihao Li. 2022. Parameter-Efficient Tuning on Layer Normalization for Pre-trained Language Models. *arXiv preprint arXiv:2211.08682* (2022).

[31] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. 2019. Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer. https://doi.org/10.48550/ARXIV.1910.10683

[32] Andreas Rücklé, Gregor Geigle, Max Glockner, Tilman Beck, Jonas Pfeiffer, Nils Reimers, and Iryna Gurevych. 2021. AdapterDrop: On the Efficiency of Adapters in Transformers. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, Online and Punta Cana, Dominican Republic, 7930–7946. https://doi.org/10.18653/v1/2021.emnlp-main.626

[33] Suraj Srinivas and R. Venkatesh Babu. 2015. Data-free parameter pruning for Deep Neural Networks. https://doi.org/10.48550/ARXIV.1507.06149

[34] Tianxiang Sun, Yunfan Shao, Hong Qian, Xuanjing Huang, and Xipeng Qiu. 2022. Black-Box Tuning for Language-Model-as-a-Service. https://doi.org/10.48550/ARXIV.2201.03514

[35] Cheng Tai, Tong Xiao, Yi Zhang, Xiaogang Wang, and Weinan E. 2015. Convolutional neural networks with low-rank regularization. https://doi.org/10.48550/ARXIV.1511.06067

[36] Karen Ullrich, Edward Meeds, and Max Welling. 2017. Soft Weight-Sharing for Neural Network Compression. https://doi.org/10.48550/ARXIV.1702.04008

[37] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention Is All You Need. https://doi.org/10.48550/ARXIV.1706.03762

[38] Alex Wang, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel R. Bowman. 2018. GLUE: A Multi-Task Benchmark and Analysis Platform for Natural Language Understanding. https://doi.org/10.48550/ARXIV.1804.07461

[39] Jiaxiang Wu, Cong Leng, Yuhang Wang, Qinghao Hu, and Jian Cheng. 2016. Quantized convolutional neural networks for mobile devices. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 4820–4828. https://doi.org/10.1109/CVPR.2016.521

[40] Runxin Xu, Fuli Luo, Zhiyuan Zhang, Chuanqi Tan, Baobao Chang, Songfang Huang, and Fei Huang. 2021. Raise a Child in Large Language Model: Towards Effective and Generalizable Fine-tuning. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, Online and Punta Cana, Dominican Republic, 9514–9528. https://doi.org/10.18653/v1/2021.emnlp-main.749