UNIPELT: A Unified Framework for Parameter-Efficient Language Model Tuning

Anonymous ACL submission

Abstract

Recent parameter-efficient language model tuning (PELT) methods manage to match the performance of fine-tuning with much fewer trainable parameters and perform especially well when training data is limited. However, different PELT methods may perform rather differently on the same task, making it nontrivial to select the most appropriate method for a specific task, especially considering the fast-growing number of new PELT methods and tasks. In light of model diversity and the difficulty of model selection, we propose a unified framework, UNIPELT, which incorporates different PELT methods as submodules and learns to activate the ones that best suit the current data or task setup via gating mechanism. On the GLUE benchmark, UNIPELT consistently achieves 1~4% gains compared to the best individual PELT method that it incorporates and even outperforms fine-tuning under different setups. Moreover, UNIPELT generally surpasses the upper bound that takes the best performance of all its submodules used individually on each task, indicating that a mixture of multiple PELT methods may be inherently more effective than single methods.[1]

1 Introduction

As pre-trained language models (PLMs) (Devlin et al., 2019) grow larger and larger (Brown et al., 2020), it becomes increasingly infeasible to perform conventional fine-tuning, where separate replicas of the model parameters are modified per single task. To solve the issue, there has recently been a surge of studies on **p**arameter-**e**fficient **l**anguage model **t**uning (PELT), namely how to effectively tune the PLMs with fewer trainable parameters.

Existing PELT research generally aims at achieving performance comparable to fine-tuning with as few trainable parameters as possible, which has seen significant progress – the task-specific trainable parameters used in most recent approaches (Lester et al., 2021; Guo et al., 2021) are almost negligible compared to the total parameters of the PLM (<1%). A more challenging yet less studied problem is whether one can achieve better performance than fine-tuning with fewer parameters. Recent studies (He et al., 2021; Li and Liang, 2021; Karimi Mahabadi et al., 2021b) find that some PELT methods could be more effective than fine-tuning on certain tasks when training data is limited, possibly due to the reduced risk of overfitting. However, as found in our experiments (Table 1), different PELT methods exhibit diverse characteristics and perform rather differently on the same task, which makes it nontrivial to select the most appropriate method for a specific task, especially

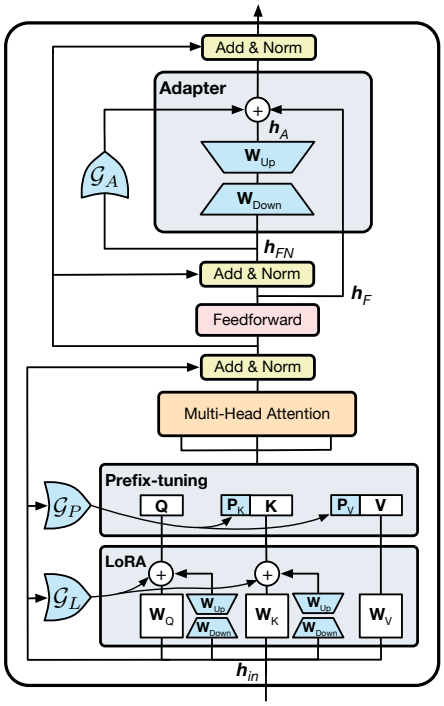

Figure 1: Illustration of UNIPELT, which subsumes existing PELT methods as submodules and controls them via gating mechanism $\mathcal{G}$. Different (combinations of) submodules can be activated for different samples. The **trainable parameters** are shown in blue.

[1] Our code will be released to facilitate future work.

considering the fast-growing number of new PELT methods and tasks (Ding and Hu, 2021).

In light of the diverse performance of PELT methods and the cost of selecting the best method, we propose a unified PELT framework, named UNIPELT, which incorporates different PELT methods as submodules and learns to dynamically activate the submodules that best suit the current data or task setup. As a result, model selection is no longer needed and consistently better performance is achieved under different setups. The activation of each submodule in UNIPELT is controlled by *gating mechanism*, which learns to favor (assign more weight to) the submodules that perform well on a given task. In addition, since the number of parameters introduced by each submodule is generally small, combining multiple methods leads to negligible losses in model efficiency.

We select four representative PELT methods for our study – adapter (Houlsby et al., 2019), prefix-tuning (Li and Liang, 2021), BitFit (Ben Zaken et al., 2021), and LoRA (Hu et al., 2021), which largely cover the major categories of PELT methods. We perform two sets of analysis that carefully examines (i) the characteristics of individual PELT methods and (ii) their effectiveness when coordinated together by UNIPELT under various setups.

Extensive experiments on the GLUE benchmark (Wang et al., 2019), with 32 task×data setups (8×4) and 1,000+ runs, not only reveal the diverse behavior of PELT methods, but also show that UNIPELT is more effective and robust than using each method alone in various task and data setups. Specifically, UNIPELT consistently improves the best submodule that it incorporates by 1~4 points and even outperforms fine-tuning, achieving the best average performance on the GLUE benchmark under different setups. Moreover, UNIPELT generally surpasses the upper bound that takes the best performance of all its submodules used individually on each task, which suggests that UNIPELT maintains (near) optimal performance under different setups. The fact that UNIPELT outperforms the upper bound also implies that a mixture of PELT methods that involve different parts of the PLM architecture may be inherently more effective than individual methods.

Contributions. (1) We conduct a comprehensive study of representative PELT methods and carefully examine their differences and commonalities in terms of performance and characteristics. (2) We propose a unified PELT framework that can incorporate existing methods as submodules and automatically learn to activate the most appropriate submodules for a given task. (3) Our proposed framework achieves better average performance than fine-tuning and the PELT methods that it incorporates under various setups, often performing the best and never the worst at per-task level, exhibiting superior effectiveness and robustness with negligible losses in model efficiency.

2 Preliminaries

2.1 PELT methods w/o Additional Parameters

PLMs can be used as feature extractors where only the top layers or prediction head are fine-tuned without additional parameters (Lee et al., 2019). However, such fine-tuning approaches generally lead to degenerate model performance that is much worse than fine-tuning all parameters (Lee et al., 2019; Pfeiffer et al., 2021). A recent method BitFit (Ben Zaken et al., 2021) only tunes the bias terms of the PLM and is shown to achieve performance comparable to fine-tuning on certain tasks when training data is limited. Therefore, we select BitFit as the representative of this category.

2.2 PELT methods w/ Additional Parameters

Alternatively, one may fix the entire PLM and introduce a small number of new trainable parameters. Notable examples in this category include adapter (Houlsby et al., 2019) and its extensions (Pfeiffer et al., 2021; Karimi Mahabadi et al., 2021b), prefix-tuning (Li and Liang, 2021) and its extensions (Lester et al., 2021), and additive methods (Guo et al., 2021; Hu et al., 2021).

Next, we will briefly describe these methods to facilitate the introduction of our proposed framework. An illustration is shown in Fig. 1 for better understanding.

Adapter. Adapter (Houlsby et al., 2019) adds a trainable *bottleneck layer* after the feedforward network in each Transformer layer of the PLM. A bottleneck layer consists of a down+up projection pair that shrinks and recovers the size of token hidden states. Mathematically, if we denote the output of the **f**eedforward network after residual connection and layer **n**ormalization as $\boldsymbol{h}_{FN}$ with hidden size D_{hidden} and bottleneck size D_{mid}, then the output of a bottleneck layer $\boldsymbol{h}_A$ is:

$$\boldsymbol{h}_A = \boldsymbol{W}_{\text{up}}^{\intercal}\phi(\boldsymbol{W}_{\text{down}}^{\intercal}\boldsymbol{h}_{FN}), \quad (1)$$

where $\boldsymbol{W}_{\text{down}} \in \mathbb{R}^{D_{\text{hidden}} \times D_{\text{mid}}}$, $\boldsymbol{W}_{\text{up}} \in \mathbb{R}^{D_{\text{mid}} \times D_{\text{hidden}}}$, ϕ is a nonlinear activation function, and the bias terms are omitted for brevity. The parameters in layer normalization and the final prediction head sometimes are also fine-tuned depending on the specific adapter variants.

Adapter has shown to be on par with fine-tuning and sometimes exhibits better effectiveness in the low-resource setting (He et al., 2021). Later studies extend adapter to multi-lingual (Pfeiffer et al., 2020b) and multi-task (Karimi Mahabadi et al., 2021b) settings, or further reduce its trainable parameters (Karimi Mahabadi et al., 2021a), which can be easily incorporated into UNIPELT as a replacement of the vanilla adapter.

Prefix-tuning. Prefix-tuning (Li and Liang, 2021) prepends a number of task-specific trainable vectors to the input of multi-head attention in *each* Transformer layer, which the original tokens can attend to as if they were virtual tokens. Specifically, we denote the original sequence length L_0, the number of trainable vectors (*i.e.*, prefix length) L, and the Transformer layer input $\boldsymbol{h}_{\text{in}} \in \mathbb{R}^{D_{\text{hidden}} \times L_0}$. First, three linear projections $\boldsymbol{W}_Q$, $\boldsymbol{W}_K$, $\boldsymbol{W}_V \in \mathbb{R}^{D_{\text{hidden}} \times D_{\text{hidden}}}$ transform $\boldsymbol{h}_{\text{in}}$ into Query $\boldsymbol{Q}$, Key $\boldsymbol{K}$, and Value $\boldsymbol{V}$. Then, two prefix matrices $\boldsymbol{P}_K$ and $\boldsymbol{P}_V \in \mathbb{R}^{D_{\text{hidden}} \times L}$ are prepended to $\boldsymbol{K}$ and $\boldsymbol{V}$. To allow for more expressiveness, the prefix matrix $\boldsymbol{P}$ is reparameterized by a feedforward network:

$$\boldsymbol{P} = \boldsymbol{W}_{\text{up}}^{\intercal} \phi(\boldsymbol{W}_{\text{down}}^{\intercal} \boldsymbol{P}), \tag{2}$$

where $\boldsymbol{W}_{\text{down}} \in \mathbb{R}^{D_{\text{hidden}} \times D_{\text{mid}}}$, $\boldsymbol{W}_{\text{up}} \in \mathbb{R}^{D_{\text{mid}} \times 2N_{\text{layer}} D_{\text{hidden}}}$, and N_{layer} denotes the number of Transformer layers. The parameters of this network can be discarded after training, and only $2N_{\text{layer}}$ prefix matrices $\in \mathbb{R}^{D_{\text{hidden}} \times L}$ are needed (2 matrices for each layer).

Prefix-tuning is originally evaluated on natural language generation and we adapt it to understanding tasks. A follow-up method named prompt-tuning (Lester et al., 2021) further reduces task-specific parameters by limiting the prefix to the first layer but only performs competitively with very large model sizes (billions of total parameters), and is thus not considered in our study. Note that prefix-tuning (or prompt-tuning) is different from prompt-based *fine-tuning* methods (Schick and Schütze, 2021; Gao et al., 2021) in multiple ways. Please see App. A for specific differences.

Additive Methods. Additive PELT methods treat the model parameters after fine-tuning as an addition of the pre-trained parameters $\theta_{\text{pre-trained}}$ and task-specific differences δ_{task}, where $\theta_{\text{pre-trained}}$ is fixed and a new (sub)set of model parameters are added on top: $\theta_{\text{task}} = \theta_{\text{pre-trained}} + \delta_{\text{task}}$. There are various ways to parameterize δ_{task}, leading to different additive methods such as LoRA (Hu et al., 2021), diff pruning (Guo et al., 2021), and side-tuning (Zhang et al., 2020). We take LoRA as a representative and incorporate it into UNIPELT. Other methods are conceptually similar and can be incorporated in the same fashion.

LoRA introduces trainable low-rank matrices and combines them with the original matrices in the multi-head attention. Specifically, two matrices $\boldsymbol{W}_{\text{down}} \in \mathbb{R}^{D_{\text{hidden}} \times D_{\text{mid}}}$ and $\boldsymbol{W}_{\text{up}} \in \mathbb{R}^{D_{\text{mid}} \times D_{\text{hidden}}}$ are added for the query and key projections along with the original matrix $\boldsymbol{W}_Q$ and $\boldsymbol{W}_K \in \mathbb{R}^{D_{\text{hidden}} \times D_{\text{hidden}}}$:

$$\boldsymbol{Q} = (\boldsymbol{W}_Q^{\intercal} + \alpha \boldsymbol{W}_{\text{up}}^{\intercal} \boldsymbol{W}_{\text{down}}^{\intercal}) \boldsymbol{h}_{\text{in}}, \tag{3}$$

where α is a fixed scalar hyperparameter for scaling the task-specific differences. The form of the trainable matrices is quite similar to those in adapter or prefix-tuning, but there is no activation function ϕ in between.

3 Unifying PELT Methods

3.1 Task Formulation

Given a large PLM $\mathcal{M}$ with size $|\mathcal{M}|$ that cannot be fine-tuned directly due to computational or storage cost, suppose that we have a list of PELT methods $\{m_i\}$, the trainable parameters of which are negligible (*i.e.*, $\sum_i |m_i| \ll |\mathcal{M}|$), our goal is to design a unified PELT framework that incorporates $\{m_i\}$ as submodules and learns to dynamically activate (up-weight) different submodules when appropriate under different scenarios, such that one could achieve satisfactory results in terms of both model effectiveness and robustness without the hassle of permuting all the method$\times$task$\times$data combinations.

3.2 Proposed Method

Motivation & Intuition. During the analysis of individual PELT methods, we observe that different PELT methods exhibit diverse characteristics and perform rather differently on the same task. For example, prefix-tuning generally performs well on natural language inference tasks regardless of the size of training data. Also, as can be seen in Fig. 1 and Sec. 2, different PELT methods often involve different parts of the PLM architecture (*e.g.*, before multi-head attention for prefix-tuning and

after feedforward layer for adapter), making it feasible to combine multiple PELT methods without (directly) interfering with each other.

In light of the two observations above, we propose a unified PELT framework, UNIPELT, which takes a hybrid approach by incorporating multiple PELT methods as submodules. At a high level, UNIPELT improves over single PELT methods due to two factors. First, UNIPELT learns to activate (upweight) the submodules that best suit the current task or specific data sample and deactivate (downweight) the rest. Second, we find that UNIPELT generally performs better than taking the best performance of all its submodules used individually on each task, suggesting that there could be some compounding effects that lead to better model effectiveness when multiple PELT methods (that modify different parts of the PLM) are used.

Next, we will introduce how different PELT methods can be incorporated into UNIPELT via gating mechanism.[2]

Gating Mechanism. To achieve fine-grained control of submodule (de)activation, we add a trainable gate $\mathcal{G}_{m_i}$ for each submodule $m_i \in \{A, P, L\}$ in every Transformer layer (see Fig. 1). The letters A, P, L stand for Adapter, Prefix-tuning, and LoRA, respectively. Intuitively, if m_i is useful for a given data or task setup, the gate output for m_i would be high such that m_i plays a more important role (although the actual interactions could be more complicated given the compounding effects of multiple layers and submodules).

Specifically, for adapter, there is a residual connection between the feedforward network and the adapter submodule that sums the adapter input (before normalization) $\boldsymbol{h}_F$ and output $\boldsymbol{h}_A$ as its final output: $\boldsymbol{h}_A = \boldsymbol{h}_A + \boldsymbol{h}_F$. We design a gating function $\mathcal{G}_A \in (0, 1)$ that estimates the importance of adapter by its direct input $\boldsymbol{h}_{FN}$ using a feedforward network with sigmoid activation and then scales its output: $\boldsymbol{h}_A = \mathcal{G}_A \boldsymbol{h}_A + \boldsymbol{h}_F$. The adapter submodule is effectively bypassed if $\mathcal{G}_A \approx 0$.

Similarly, for prefix-tuning, we design a gating function $\mathcal{G}_P \in (0, 1)$ that is applied to the prefix vectors ($\boldsymbol{P}_K$ and $\boldsymbol{P}_V$) with the representation of the original tokens ($\boldsymbol{K}$ and $\boldsymbol{V}$) intact. In this way, the impact of the prefix would be diminished if the gate output of the prefix-tuning submodule is low.[3] The gating function $\mathcal{G}_P$ is estimated by the Transformer layer input $\boldsymbol{h}_{\text{in}}$ with another feedforward network.

As for LoRA, we note that there is already a *constant* scaling factor α in its original design that resembles the purpose of our gating mechanism. We thus simply make the factor learnable per layer by a third feedforward network that takes $\boldsymbol{h}_{\text{in}}$ as input instead of specifying a constant manually: $\theta_{\text{task}} = \theta_{\text{pre-trained}} + \mathcal{G}_L \delta_{\text{task}}$.

Despite the seeming simplicity of UNIPELT, we note that it is nontrivial for a unified approach to work well under different scenarios. Naively combining different PELT methods as a hybrid approach may lead to mixed or worse performance than using individual methods, as observed in both our experiments and prior studies (Hu et al., 2021).

4 Experiments

We conduct extensive experiments with 8 tasks × 4 data sizes × 7 methods × 5 runs per setup, along with additional analysis for particular methods, resulting in 1,000+ runs in total.

4.1 Experiment Setup

Task Setup. We conduct experiments on the General Language Understanding Evaluation (GLUE) benchmark (Wang et al., 2019), which involves four types of natural language understanding tasks including linguistic acceptability (CoLA), sentiment analysis (SST-2), similarity and paraphrase tasks (MRPC, STS-B, QQP), and natural language inference (MNLI, QNLI, RTE). WNLI is omitted following prior studies (Houlsby et al., 2019; Devlin et al., 2019; He et al., 2021; Ben Zaken et al., 2021) due to its adversarial nature.

Data Setup. We mainly consider a low-resource setting where training data is limited and the performance of different methods varies much. We sample a small subset of the training set for each task with size $K = \{100, 500, 1000\}$. As it is infeasible to submit considerable runs to the GLUE leaderboard (2 submissions/day), we take 1,000 samples on the training set as the development set to select the best checkpoint and use the original development set as the test set (He et al., 2021). To reduce variance, we shuffle the data with 5 random seeds and report the average performance. Additionally, we consider a high-resource setting where the whole training set is used and the best performance on the GLUE development set is reported.

Compared Methods. We mainly compare

[2] BitFit is not included in UNIPELT as it does not perform very well in our experiments.

[3] Prefix-tuning cannot be fully eliminated as adapter or LoRA due to the softmax operation in multi-head attention.

Method	SST-2	MRPC	CoLA	RTE	QNLI	STS-B	MNLI	QQP	Avg.
[$K = 100$] Test Performance									
Fine-tuning	$79.61_{4.25}$	$\underline{81.81}_{0.35}$	$\underline{16.56}_{4.34}$	$\underline{55.88}_{1.64}$	$69.25_{5.94}$	$74.07_{6.51}$	$\underline{42.56}_{3.43}$	$60.41_{6.42}$	$60.02_{1.84}$
BitFit	$62.94_{4.85}$	$81.09_{0.17}$	$2.71_{1.57}$	$47.65_{3.56}$	$42.46_{1.37}$	$54.53_{0.56}$	$38.16_{0.53}$	$59.56_{0.39}$	$48.64_{0.78}$
Adapter	$80.48_{2.94}$	$81.40_{0.19}$	$2.02_{4.04}$	$52.78_{0.27}$	$72.25_{0.49}$	$77.32_{1.54}$	$38.81_{3.64}$	$60.88_{4.00}$	$58.24_{0.99}$
Prefix-tuning	$60.87_{12.47}$	$81.22_{0.00}$	$0.00_{0.00}$	$\mathbf{55.96}_{2.00}$	$71.91_{2.69}$	$57.69_{0.02}$	$40.58_{2.49}$	$15.68_{0.12}$	$47.99_{1.77}$
$\rightarrow L = 50$	$79.52_{1.21}$	$81.22_{0.00}$	$5.19_{8.62}$	$49.24_{2.08}$	$66.33_{2.45}$	$7.15_{10.37}$	$33.66_{2.21}$	$58.32_{3.18}$	$47.56_{1.37}$
LoRA	$\underline{81.56}_{0.94}$	$81.66_{0.81}$	$13.31_{10.00}$	$55.02_{1.75}$	$\mathbf{73.52}_{1.20}$	$49.35_{21.87}$	$39.60_{4.98}$	$0.09_{0.02}$	$49.26_{2.19}$
UNIPELT (AP)	$77.22_{3.75}$	$\mathbf{81.86}_{0.70}$	$14.42_{10.24}$	$55.52_{2.16}$	$72.26_{0.89}$	$\underline{79.14}_{1.97}$	$\mathbf{42.59}_{1.20}$	$\mathbf{63.41}_{1.44}$	$\underline{60.80}_{1.53}$
UNIPELT (APL)	$\mathbf{82.36}_{0.86}$	$81.71_{0.72}$	$\mathbf{23.62}_{8.83}$	$55.45_{1.28}$	$\underline{73.19}_{0.93}$	$\mathbf{79.37}_{1.07}$	$42.30_{1.88}$	$\underline{62.70}_{2.55}$	$\mathbf{62.59}_{1.44}$
[$K = 500$] Test Performance									
Fine-tuning	$\mathbf{85.67}_{0.97}$	$\underline{83.34}_{0.55}$	$36.47_{2.69}$	$59.64_{1.10}$	$77.30_{0.49}$	$\underline{84.96}_{1.19}$	$55.84_{0.85}$	$\underline{68.23}_{1.39}$	$68.93_{0.65}$
BitFit	$83.44_{0.63}$	$82.16_{0.37}$	$3.32_{2.59}$	$61.88_{2.75}$	$69.15_{9.91}$	$76.30_{0.36}$	$40.82_{3.30}$	$65.29_{3.66}$	$60.30_{1.91}$
Adapter	$84.54_{1.37}$	$82.53_{0.36}$	$38.65_{3.97}$	$59.35_{3.09}$	$77.39_{0.84}$	$83.52_{0.33}$	$50.04_{1.72}$	$68.12_{0.95}$	$68.02_{0.77}$
Prefix-tuning	$83.65_{0.69}$	$82.96_{1.63}$	$38.16_{2.25}$	$63.18_{2.70}$	$78.50_{1.12}$	$79.75_{1.49}$	$\mathbf{58.06}_{1.04}$	$54.34_{25.91}$	$67.32_{3.42}$
LoRA	$\underline{84.98}_{1.10}$	$82.53_{0.70}$	$\mathbf{39.86}_{2.71}$	$63.03_{2.57}$	$\mathbf{79.46}_{0.66}$	$65.05_{26.31}$	$\underline{56.54}_{2.05}$	$55.46_{27.74}$	$65.86_{4.18}$
UNIPELT (AP)	$84.84_{0.28}$	$83.25_{0.51}$	$\underline{39.84}_{5.01}$	$\underline{63.32}_{1.72}$	$78.36_{1.06}$	$84.53_{0.48}$	$56.08_{3.26}$	$68.14_{1.39}$	$\underline{69.79}_{1.02}$
UNIPELT (APL)	$84.91_{1.41}$	$\mathbf{83.56}_{0.59}$	$39.81_{2.55}$	$\mathbf{64.12}_{2.45}$	$\underline{79.28}_{0.63}$	$\mathbf{85.26}_{0.70}$	$54.07_{3.74}$	$\mathbf{68.87}_{0.41}$	$\mathbf{69.98}_{0.42}$
[$K = 1000$] Test Performance									
Fine-tuning	$\underline{86.54}_{1.01}$	$84.87_{0.64}$	$43.26_{2.60}$	$62.31_{2.10}$	$79.03_{1.11}$	$86.39_{0.34}$	$61.95_{1.20}$	$\mathbf{71.09}_{0.77}$	$71.93_{0.37}$
BitFit	$83.99_{0.39}$	$83.95_{0.81}$	$22.44_{17.10}$	$62.89_{1.40}$	$77.43_{0.53}$	$79.04_{0.61}$	$52.87_{0.72}$	$69.50_{0.16}$	$66.51_{2.22}$
Adapter	$85.60_{0.63}$	$84.49_{0.60}$	$42.33_{1.98}$	$61.81_{1.57}$	$79.68_{0.23}$	$85.52_{0.29}$	$57.86_{2.44}$	$70.32_{0.71}$	$70.95_{0.55}$
Prefix-tuning	$85.09_{0.99}$	$83.66_{1.82}$	$44.07_{2.90}$	$\mathbf{66.71}_{2.72}$	$80.34_{0.70}$	$82.38_{1.25}$	$\mathbf{63.59}_{1.12}$	$68.58_{0.35}$	$71.81_{0.52}$
LoRA	$86.26_{1.22}$	$\underline{86.04}_{0.99}$	$\mathbf{45.50}_{1.11}$	$\underline{65.63}_{2.11}$	$\underline{81.00}_{0.98}$	$81.56_{1.97}$	$61.32_{1.65}$	$70.89_{0.81}$	$72.28_{0.69}$
UNIPELT (AP)	$86.17_{0.37}$	$85.86_{1.05}$	$44.33_{3.55}$	$64.91_{1.92}$	$80.65_{0.57}$	$\underline{86.82}_{0.23}$	$62.17_{0.99}$	$69.95_{0.90}$	$\underline{72.61}_{0.53}$
UNIPELT (APL)	$\mathbf{87.06}_{0.81}$	$\mathbf{86.65}_{1.10}$	$\underline{45.44}_{1.97}$	$65.49_{1.92}$	$\mathbf{81.22}_{0.51}$	$\mathbf{87.10}_{0.21}$	$\underline{62.49}_{0.94}$	$\underline{70.99}_{0.95}$	$\mathbf{73.31}_{0.52}$

Table 1: Results on the GLUE benchmark with $K = \{100, 500, 1000\}$ training samples. The evaluation metrics are Matthew's Correlation for CoLA, F1 for MRPC and QQP, Spearman's correlation for STS-B, and accuracy for the rest. For MNLI, we evaluate on the matched dataset. We report average performance on five random seeds with standard deviation as the subscript. **Best** and 2nd best methods under each setup are bold and underlined.

UNIPELT with fine-tuning and four representative PELT methods: adapter (Houlsby et al., 2019), prefix-tuning (Li and Liang, 2021), BitFit (Ben Zaken et al., 2021), and LoRA (Hu et al., 2021). For completeness, we consider two model variants UNIPELT (AP) and UNIPELT (APL), which incorporate 2 and 3 PELT methods, respectively.

Implementation Details. We use $\text{BERT}_{\text{base}}$ as the base model in the experiments. We implement and evaluate all the methods in the same codebase to ensure a fair comparison. We largely follow the default hyperparameters of different methods and keep them the same on all the tasks for generalizability. We set the prefix length $L = 10$, adapter bottleneck size $D_{\text{mid}} = 48$, LoRA rank $D_{\text{mid}} = 8$ if not specified otherwise.[4] More implementation and hyperparameter details can be found in App. B.

4.2 Analysis of Individual PELT Methods

In Table 1, we show the performance of different methods on the GLUE benchmark with various sizes of training data. The results on the development sets are generally consistent with the test sets and provided in App. C due to space limit. As one can see, although the average performance of different methods over 8 tasks is sometimes similar, the differences are quite significant under certain setups and can be as large as 5~9 points on a specific task (*e.g.*, STS-B and MNLI, $K = 500$) even when excluding cases where some methods fail to learn effectively (*e.g.*, prefix-tuning on QQP, $K = 100$).

Next, we will analyze and examine each individual PELT method more closely.

Analysis of Adapter. The performance of adapter is relatively stable – there is no significantly better or worse result than fine-tuning consistent on different tasks or sizes of training data. In general, adapter is slightly worse than fine-tuning in most cases. We do not observe that adapter consistently outperforms fine-tuning in the low-resource setting as in He et al. (2021), possibly because they tune model hyperparameters on each task, which could be computationally prohibitive when there are considerable tasks. For example, they tune the bottleneck size D_{mid} from {64, 128, 256}, while $D_{\text{mid}} = 48$ is fixed across tasks in our experiments.

[4] While these hyperparameters may lead to differences in trainable parameters, we keep them for analysis as they are used by the official implementation. Also, we observe that more trainable parameters do not guarantee better results.

Also, we only add one adapter in each Transformer layer instead of two following Pfeiffer et al. (2021). These two differences result in 62.4%~90.5% fewer parameters than the adapter used in He et al. (2021).

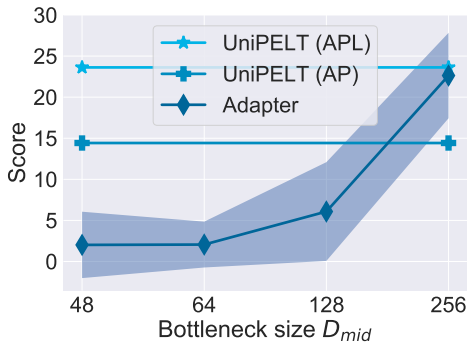

Figure 2: Performance changes when the bottleneck size of adapter is increased (on CoLA, $K = 100$).

To further study the effect of bottleneck size D_{mid} in adapter, we increase D_{mid} and re-evaluate adapter on a setup that it performs poorly (CoLA, $K = 100$). As shown in Fig. 2, the performance of adapter is increased gradually and becomes significantly better only when $D_{\text{mid}} = 256$, which involves 5.3× trainable parameters than the adapter used originally ($D_{\text{mid}} = 48$), 4.3× than UniPELT (AP), and 3.4× than UniPELT (APL), suggesting that a larger bottleneck size could be beneficial when adapter learns ineffectively.

On the other hand, there are certain tasks (*e.g.*, STS-B) that adapter largely outperforms competitive methods such as prefix-tuning and LoRA regardless of the size of training data, suggesting that one should favor adapter over other PELT methods under certain scenarios as well.

Analysis of Prefix-tuning. Prefix-tuning performs poorly with $K = \{100, 500\}$ and becomes on par with fine-tuning when K reaches 1000. We also observe that prefix-tuning fails to learn effectively on certain tasks when the training data is limited (*e.g.*, $K = 100$ on SST-2 and $K = 500$ on QQP), leading to unsatisfactory performance and (or) large variance across different runs. Similar phenomena have been observed in a concurrent study (Gu et al., 2021) on few-shot prompt-tuning.

To ensure that the poor performance of prefix-tuning is not due to its fewer trainable parameters (based on its default setting), we further increase the prefix length to $L = 50$ such that its trainable parameters are comparable to adapter, and re-evaluate prefix-tuning on all 8 tasks with $K = 100$. For the 4 tasks where prefix-tuning ($L = 10$) performs poorly (SST2, CoLA, STS-B, and QQP), while its performance is significantly improved on 3 tasks, it also performs significantly worse on the other task (STS-B), which suggests that training instability in the low-resource regime is still an issue for prefix-tuning even with more trainable parameters.[5] Besides, prefix-tuning ($L = 50$) still lags behind adapter or UniPELT (AP) on 3 of the 4 tasks. Furthermore, the average performance of prefix-tuning ($L = 50$) on 8 tasks is even slightly worse than with $L = 10$, which indicates that increasing prefix length may not be a panacea for all the scenarios. A larger L also leads to significant training/inference slowdown due to the costly multi-head attention. More broadly, such results suggest that using more trainable parameters does not guarantee better performance.

On the bright side, prefix-tuning performs especially well on certain tasks such as natural language inference (QNLI and MNLI) with various sizes of training data, which suggests that a hybrid approach that learns to activate prefix-tuning on these tasks is likely to yield decent results.

Analysis of BitFit & LoRA. Tuning only the bias terms of the model does not lead to very satisfactory results in our experiments – BitFit never performs the best and generally performs the worst in different data and task setups. Therefore, we do not consider BitFit in the following experiments and exclude BitFit as a submodule of UniPELT. As for LoRA, There are a few setups where LoRA fails to learn effectively as well, such as STS-B and QQP ($K = \{100, 500\}$), leading to high variance across runs. Apart from that, LoRA performs quite competitively despite using fewer trainable parameters than methods like adapter, especially when $K = 1000$, achieving the best or 2nd best performance on 4 of 8 tasks.

As LoRA has a scaling factor α that can be seen as a static gating function under our formulation, we further investigate its importance by evaluating LoRA with different α. As shown in Fig. 3, LoRA is quite sensitive to the scaling factor and there seems to be no single optimal value that works well across multiple task and data setups. Such findings suggest that gating is critical and motivate us to use more fine-grained and dynamic control for UniPELT. Besides, we observe that increasing α consistently results in faster convergence, possibly because the trainable parameters would receive

[5] Tuning other hyperparameters like learning rate does not appear to alleviate the issue either.

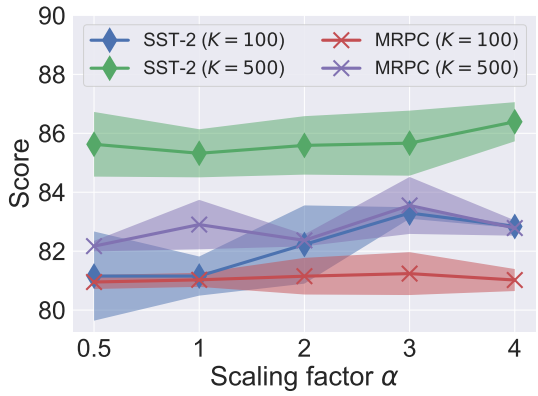

Figure 3: Performance comparison of various scaling factors for LoRA on 2×2 task and data setups.

larger gradient updates with a larger α.

4.3 Analysis of UNIPELT

Next, we will turn to our proposed framework UNIPELT, which incorporates multiple existing PELT methods as submodules.

Low-Resource Performance. Overall, UNIPELT (APL) and UNIPELT (AP) consistently achieve the best and 2nd best average performance on both the development and test sets regardless of the number of training samples. The gains are generally 1~4% over the submodule that performs the best (when used individually). Such results demonstrate the advantages of our hybrid approach regarding model effectiveness and generalizability.

At the per-task level, UNIPELT (APL) and UNIPELT (AP) perform the best or 2nd best on 7/6/7 of 8 tasks when trained with 100/500/1,000 samples, and never perform the worst in any setup. When comparing the two variants, UNIPELT (APL) outperforms UNIPELT (AP) on 4/6/8 of 8 tasks when trained with 100/500/1,000 samples. Such results indicate that UNIPELT is quite robust and performs reliably under different scenarios. The improvements of UNIPELT over its submodules are generally larger when having fewer training samples, suggesting that UNIPELT performs especially well in the low-resource regime. In particular, on the tasks where other PELT methods fail to learn effectively such as CoLA and QQP ($K = 100$), UNIPELT manages to achieve performance better than fine-tuning.

UNIPELT vs. Upper Bound. In Table 2, we show the comparison of UNIPELT and the upper bound that takes the best performance of its submodules on each task. We observe that both UNIPELT (AP) and UNIPELT (APL) perform similarly or even better than their upper bound, which suggests that UNIPELT successfully learns to leverage different submodules and maintains (near) optimal performance under different setups. The fact that UNIPELT can outperform the upper bound also hints that a mixture of PELT methods (involving different parts of the PLM) might be inherently more effective than single methods (with a limited scope of the PLM architecture).

K	max($\{A, P\}$)	UNIPELT	max($\{A, P, L\}$)	UNIPELT
100	58.86	**60.80**	60.60	**62.59**
500	69.69	**69.79**	**70.02**	69.98
1000	72.58	**72.61**	73.19	**73.31**

Table 2: Comparison of average test performance between UNIPELT and the upper bound that takes the best performance of its submodules on each task.

High-Resource Performance. In Table 3, we list the performance of different methods when all training samples are used. UNIPELT again achieves the best overall performance. The gains are not as significant as in the low-resource setting, which is somewhat expected as existing PELT methods typically perform on par with fine-tuning given abundant training data and the potential of improvement is not as high. That said, the performance of UNIPELT is still the best or 2nd best on all 8 tasks, and generally comparable to the best submodule used individually on each task. Besides, simply combining multiple PELT methods without gating may not work very well – although UNIPELT-NoGate never performs the worst in each task, its overall performance is unsatisfactory (-0.89), which suggests that a more careful mixture of PELT methods is important for achieving better model effectiveness.

4.4 Efficiency of PELT Methods

We benchmark the efficiency of PELT methods and list in Table 4 their number of trainable parameters and training/inference time relative to fine-tuning.

Parameter Efficiency. As the trainable parameters in PELT methods are almost negligible, combining multiple methods does not lead to significant losses in parameter efficiency. UNIPELT still has few trainable parameters compared to fine-tuning (0.99%~1.26%). The number can be further reduced (to *e.g.*, <0.1%) if one uses more parameter-efficient variants, which can be easily swapped with the vanilla version used in our current framework. Also, note that more trainable parameters do not always lead to better performance, as shown in

Method	SST-2	MRPC	CoLA	RTE	QNLI	STS-B	MNLI	QQP	Avg.
[$K = all$] Best Performance on GLUE Dev									
Fine-tuning	91.63	90.94	**62.08**	66.43	89.95	**89.76**	83.23	**87.35**	82.67
Adapter	**91.86**	89.86	61.51	71.84	90.55	88.63	83.14	86.78	83.02
Prefix-tuning	90.94	**91.29**	55.37	**76.90**	90.39	87.19	81.15	83.30	82.07
LoRA	91.51	90.03	60.47	71.48	89.93	85.65	82.51	85.98	82.20
UniPELT (AP)	**91.86**	90.28	61.15	71.84	**90.77**	88.86	83.41	86.74	83.12
-NoGate	91.74	90.18	58.63	71.12	90.30	88.76	81.58	85.53	82.23
UniPELT (APL)	91.51	90.94	61.53	73.65	90.50	88.93	**83.89**	87.12	**83.50**

Table 3: Results on the GLUE benchmark when all training samples are used.

Method	#Param.	Time$_T$	Time$_I$
Fine-tuning	110M (100%)	100%	100%
BitFit	103K (0.09%)	65%	102%
Prefix-tuning	184K (0.17%)	56%	114%
LoRA	295K (0.27%)	53%	105%
Adapter	895K (0.81%)	55%	107%
UniPELT (AP)	1.1M (0.99%)	55%	118%
UniPELT (APL)	1.4M (1.26%)	67%	127%

Table 4: Number of trainable parameters and Training/Inference time relative to fine-tuning.

our experiments and prior studies (He et al., 2021; Pfeiffer et al., 2021).

Training and Inference Efficiency. Due to parameter efficiency, all PELT methods train 30%~50% faster than fine-tuning and incorporating multiple PELT methods into UniPELT does not suffer from slower training. On the other hand, the inference time of PELT methods is generally longer since they involve more FLOPs. UniPELT has a slightly larger inference overhead (4%~11% compared to its slowest submodule), which we argue is insignificant since larger models that may achieve similar performance gains (*e.g.*, BERT$_{\text{large}}$) need around 300% inference time (Wolf et al., 2020).

5 Related Work

Parameter-Efficient Tuning of PLMs. As it is increasingly infeasible to train and store full copies of large PLMs for various downstream tasks, how to efficiently tune the PLMs with few trainable parameters becomes critical. Existing PELT methods can be largely divided into two categories based on whether new trainable parameters are introduced. Specifically, one may either train a subset of the model parameters such as the prediction head (Lee et al., 2019) and bias terms (Ben Zaken et al., 2021), or introduce task-specific parameters to different parts of the PLM such as before multi-head attention (Li and Liang, 2021) or after feedforward layer (Houlsby et al., 2019). As the number of PELT methods keeps increasing, the purpose of UniPELT is to better understand and leverage the distinctions of various methods instead of proposing yet another method.

Mixture-of-Experts. UniPELT is also related to approaches that involve a high-capacity network and activate (upweight) different parts of the network given different inputs. One notable example is Mixture-of-Experts (MoE) (Shazeer et al., 2017; Hazimeh et al., 2021), which maintains a set of experts (neural networks) and one or more trainable gates that select a combination of the experts specific to each input. Despite being conceptually similar, UniPELT is different from MoE: the submodules in UniPELT are not combined explicitly by summation like MoE but in sequential order and affect each other implicitly. Moreover, the "experts" are diverse in UniPELT while usually homogeneous or identical in MoE methods.

6 Conclusion

In this paper, we present a comprehensive study of representative parameter-efficient language model tuning (PELT) methods and propose a unified framework, which incorporates different PELT methods as submodules and learns to activate the most appropriate submodules for a given task or data setup. Our proposed framework consistently outperforms conventional fine-tuning as well as the submodules that it incorporates under different setups, and generally surpasses the upper bound that takes the best performance of each submodule used individually on each task. Our findings suggest that a mixture of multiple PELT methods that involve different parts of the PLM may be favorable regarding both model effectiveness and robustness. For future work, we will try to better understand the discrepancy of various PELT methods in different scenarios. We also plan to investigate a multi-task setting where multiple submodules can be activated and cooperate at the task level.

References

Elad Ben Zaken, Shauli Ravfogel, and Yoav Goldberg. 2021. Bitfit: Simple parameter-efficient fine-tuning for transformer-based masked language-models. *arXiv e-prints*, pages arXiv–2106.

Tom B Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186, Minneapolis, Minnesota. Association for Computational Linguistics.

Ning Ding and Shengding Hu. 2021. Must-read papers on prompt-based tuning for pre-trained language models. https://github.com/thunlp/PromptPapers.

Tianyu Gao, Adam Fisch, and Danqi Chen. 2021. Making pre-trained language models better few-shot learners. 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)*, pages 3816–3830, Online. Association for Computational Linguistics.

Yuxian Gu, Xu Han, Zhiyuan Liu, and Minlie Huang. 2021. Ppt: Pre-trained prompt tuning for few-shot learning. *arXiv preprint arXiv:2109.04332*.

Demi Guo, Alexander Rush, and Yoon Kim. 2021. Parameter-efficient transfer learning with diff pruning. 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)*, pages 4884–4896, Online. Association for Computational Linguistics.

Hussein Hazimeh, Zhe Zhao, Aakanksha Chowdhery, Maheswaran Sathiamoorthy, Yihua Chen, Rahul Mazumder, Lichan Hong, and Ed H Chi. 2021. Dselect-k: Differentiable selection in the mixture of experts with applications to multi-task learning. *arXiv preprint arXiv:2106.03760*.

Ruidan He, Linlin Liu, Hai Ye, Qingyu Tan, Bosheng Ding, Liying Cheng, Jiawei Low, Lidong Bing, and Luo Si. 2021. On the effectiveness of adapter-based tuning for pretrained language model adaptation. 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)*, pages 2208–2222, Online. Association for Computational Linguistics.

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 *International Conference on Machine Learning*, pages 2790–2799. PMLR.

Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, and Weizhu Chen. 2021. Lora: Low-rank adaptation of large language models. *arXiv preprint arXiv:2106.09685*.

Rabeeh Karimi Mahabadi, James Henderson, and Sebastian Ruder. 2021a. Compacter: Efficient low-rank hypercomplex adapter layers. *arXiv preprint arXiv:2106.04647*.

Rabeeh Karimi Mahabadi, Sebastian Ruder, Mostafa Dehghani, and James Henderson. 2021b. Parameter-efficient multi-task fine-tuning for transformers via shared hypernetworks. 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)*, pages 565–576, Online. Association for Computational Linguistics.

Jaejun Lee, Raphael Tang, and Jimmy Lin. 2019. What would elsa do? freezing layers during transformer fine-tuning. *arXiv preprint arXiv:1911.03090*.

Brian Lester, Rami Al-Rfou, and Noah Constant. 2021. The power of scale for parameter-efficient prompt tuning. *arXiv preprint arXiv:2104.08691*.

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)*, pages 4582–4597, Online. Association for Computational Linguistics.

Jonas Pfeiffer, Aishwarya Kamath, Andreas Rücklé, Kyunghyun Cho, and Iryna Gurevych. 2021. AdapterFusion: Non-destructive task composition for transfer learning. In *Proceedings of the 16th Conference of the European Chapter of the Association for Computational Linguistics: Main Volume*, pages 487–503, Online. Association for Computational Linguistics.

Jonas Pfeiffer, Andreas Rücklé, Clifton Poth, Aishwarya Kamath, Ivan Vulić, Sebastian Ruder, Kyunghyun Cho, and Iryna Gurevych. 2020a. Adapterhub: A framework for adapting transformers. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP 2020): Systems Demonstrations*, pages 46–54, Online. Association for Computational Linguistics.

Jonas Pfeiffer, Ivan Vulić, Iryna Gurevych, and Sebastian Ruder. 2020b. MAD-X: An Adapter-Based Framework for Multi-Task Cross-Lingual Transfer. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 7654–7673, Online. Association for Computational Linguistics.

Timo Schick and Hinrich Schütze. 2021. Exploiting cloze-questions for few-shot text classification and natural language inference. In *Proceedings of the 16th Conference of the European Chapter of the Association for Computational Linguistics: Main Volume*, pages 255–269, Online. Association for Computational Linguistics.

Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc Le, Geoffrey Hinton, and Jeff Dean. 2017. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. *arXiv preprint arXiv:1701.06538*.

Alex Wang, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel R Bowman. 2019. Glue: A multi-task benchmark and analysis platform for natural language understanding. In *International Conference on Learning Representations*.

Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, et al. 2019. Huggingface's transformers: State-of-the-art natural language processing. *arXiv preprint arXiv:1910.03771*.

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. 2020. How to benchmark transformer models. https://huggingface.co/transformers/benchmarks.html.

Jeffrey O Zhang, Alexander Sax, Amir Zamir, Leonidas Guibas, and Jitendra Malik. 2020. Side-tuning: A baseline for network adaptation via additive side networks. In *Computer Vision–ECCV 2020: 16th European Conference, Glasgow, UK, August 23–28, 2020, Proceedings, Part III 16*, pages 698–714. Springer.

A Prefix-tuning vs. Prompt-based Fine-tuning

We note that prefix-tuning (or prompt-tuning) is different from prompt-based *fine-tuning* methods (Schick and Schütze, 2021; Gao et al., 2021) in many ways: (1) Prompt-based fine-tuning is not parameter-efficient as it updates all model parameters while prefix-tuning only updates the prefix matrix $\boldsymbol{P}$. (2) The prompts are only used in model input for prompt-based fine-tuning but added to *every* Transformer layer in prefix-tuning (stored as different vectors). (3) Prompt-based fine-tuning typically leverages carefully designed *natural language* prompts while prefix-tuning uses continuous prompts (virtual tokens).

B Implementation Details

Data Preparation. We shuffle the training set with seed s, take the first K samples as the new training set, and the next 1,000 samples as the development set. We use $s = \{111, 222, 333, 444, 555\}$ as the data seeds and the same seed ($s = 42$) for model training. We also conduct another set of preliminary experiments by fixing the data and using 5 different random seeds for model training, the results of which are similar.

Hyperparameters. We adopt AdapterHub (Pfeiffer et al., 2020a), a library based on HuggingFace Transformers (Wolf et al., 2019), as our codebase. We largely follow the recommended hyperparameters used in different methods for a fair comparison. We set the input length to 128 and the training batch size to 16. We set the number of epochs to 50 and adopt early stopping with a patience of 10 non-increasing epochs. We set the learning rate of fine-tuning and adapter to 2e-5 and 1e-4 according to the findings in prior studies (Pfeiffer et al., 2020a; He et al., 2021). For prefix-tuning and UniPELT, as they are not previously evaluated on NLU tasks, we tune their learning rates from {1e-4, 2e-4, 5e-4} on the development set and set to 2e-4 and 5e-4, respectively. For BitFit and LoRA, we choose the learning rates commonly used in their own experiments (1e-4 and 5e-4, respectively). We set $\alpha = 2$ and $r = 8$ in LoRA according to its official scripts.

C Detailed Performance

In Table 5, we list the detailed results on both development and test sets of the GLUE benchmark. The observations and findings are largely consistent on the two evaluation splits.

Method	SST-2	MRPC	CoLA	RTE	QNLI	STS-B	MNLI	QQP	Avg.
[$K = 100$] Dev Performance									
Fine-tuning	$81.24_{0.98}$	$\mathbf{81.46}_{0.78}$	$\underline{16.94}_{2.38}$	$\mathbf{58.08}_{1.63}$	$69.66_{5.03}$	$60.64_{6.97}$	$\underline{43.18}_{3.13}$	$61.63_{6.30}$	$59.10_{1.87}$
BitFit	$62.06_{4.62}$	$80.66_{0.39}$	$5.73_{1.46}$	$50.26_{0.91}$	$42.02_{2.29}$	$31.18_{2.47}$	$38.40_{0.84}$	$61.55_{0.52}$	$46.48_{0.66}$
Adapter	$80.60_{0.85}$	$\underline{81.11}_{0.78}$	$2.19_{4.38}$	$53.16_{1.99}$	$72.58_{0.66}$	$66.00_{3.66}$	$40.30_{2.82}$	$62.32_{3.20}$	$57.28_{0.60}$
Prefix-tuning	$66.24_{12.03}$	$80.51_{0.31}$	$0.00_{0.00}$	$56.60_{1.25}$	$71.94_{2.58}$	$42.81_{1.93}$	$42.26_{1.89}$	$15.14_{0.95}$	$46.94_{1.43}$
LoRA	$\underline{82.54}_{0.84}$	$80.82_{0.50}$	$14.38_{8.57}$	$56.62_{2.01}$	$\mathbf{74.26}_{0.89}$	$47.87_{14.05}$	$41.38_{4.59}$	$0.00_{0.00}$	$49.73_{1.29}$
UNIPELT (AP)	$80.40_{1.95}$	$81.02_{0.54}$	$15.07_{6.46}$	$57.68_{1.63}$	$73.50_{0.54}$	$\underline{68.19}_{3.97}$	$\mathbf{44.50}_{1.11}$	$\mathbf{64.89}_{0.86}$	$\underline{60.66}_{1.16}$
UNIPELT (APL)	$\mathbf{83.08}_{0.54}$	$81.08_{0.53}$	$\mathbf{23.52}_{5.71}$	$\underline{57.96}_{1.49}$	$\underline{74.00}_{0.46}$	$\mathbf{68.29}_{3.01}$	$43.10_{1.13}$	$\underline{63.41}_{2.93}$	$\mathbf{61.80}_{0.77}$
[$K = 100$] Test Performance									
Fine-tuning	$79.61_{4.25}$	$\underline{81.81}_{0.35}$	$\underline{16.56}_{4.34}$	$\underline{55.88}_{1.64}$	$69.25_{5.94}$	$74.07_{6.51}$	$\underline{42.56}_{3.43}$	$60.41_{6.42}$	$60.02_{1.84}$
BitFit	$62.94_{4.85}$	$81.09_{0.17}$	$2.71_{1.57}$	$47.65_{3.56}$	$42.46_{1.37}$	$54.53_{0.56}$	$38.16_{0.53}$	$59.56_{0.39}$	$48.64_{0.78}$
Adapter	$80.48_{2.94}$	$81.40_{0.19}$	$2.02_{4.04}$	$52.78_{0.27}$	$72.25_{0.49}$	$77.32_{1.54}$	$38.81_{3.64}$	$60.88_{4.00}$	$58.24_{0.99}$
Prefix-tuning	$60.87_{12.47}$	$81.22_{0.00}$	$0.00_{0.00}$	$\mathbf{55.96}_{2.00}$	$71.91_{2.69}$	$57.69_{0.02}$	$40.58_{2.49}$	$15.68_{0.12}$	$47.99_{1.77}$
$\rightarrow L = 50$	$79.52_{1.21}$	$81.22_{0.00}$	$5.19_{8.62}$	$49.24_{2.08}$	$66.33_{2.45}$	$7.15_{10.37}$	$33.66_{2.21}$	$58.32_{3.18}$	$47.56_{1.37}$
LoRA	$\underline{81.56}_{0.94}$	$81.66_{0.81}$	$13.31_{10.00}$	$55.02_{1.75}$	$\mathbf{73.52}_{1.20}$	$49.35_{21.87}$	$39.60_{4.98}$	$0.09_{0.02}$	$49.26_{2.19}$
UNIPELT (AP)	$77.22_{3.75}$	$\mathbf{81.86}_{0.70}$	$14.42_{10.24}$	$55.52_{2.16}$	$72.26_{0.89}$	$\underline{79.14}_{1.97}$	$\mathbf{42.59}_{1.20}$	$\mathbf{63.41}_{1.44}$	$\underline{60.80}_{1.53}$
UNIPELT (APL)	$\mathbf{82.36}_{0.86}$	$81.71_{0.72}$	$\mathbf{23.62}_{8.83}$	$55.45_{1.28}$	$\underline{73.19}_{0.93}$	$\mathbf{79.37}_{1.07}$	$42.30_{1.88}$	$\underline{62.70}_{2.55}$	$\mathbf{62.59}_{1.44}$
[$K = 500$] Dev Performance									
Fine-tuning	$\underline{86.66}_{1.40}$	$82.56_{0.88}$	$37.47_{3.06}$	$62.88_{1.79}$	$77.58_{1.64}$	$77.34_{2.03}$	$58.50_{1.53}$	$\underline{69.40}_{1.32}$	$69.05_{0.38}$
BitFit	$84.66_{1.28}$	$81.80_{0.96}$	$5.66_{1.87}$	$61.88_{0.95}$	$69.32_{8.90}$	$59.55_{1.41}$	$42.62_{3.23}$	$66.06_{2.99}$	$58.94_{1.65}$
Adapter	$85.74_{1.03}$	$82.74_{0.87}$	$38.22_{4.14}$	$63.52_{1.98}$	$78.20_{1.64}$	$76.15_{1.18}$	$51.30_{2.65}$	$69.23_{1.30}$	$68.14_{0.66}$
Prefix-tuning	$\mathbf{86.72}_{1.46}$	$82.26_{1.16}$	$40.25_{5.45}$	$\mathbf{66.08}_{0.83}$	$78.44_{1.48}$	$71.41_{2.30}$	$\mathbf{60.70}_{1.47}$	$54.47_{25.86}$	$67.54_{3.45}$
LoRA	$86.36_{1.37}$	$82.38_{1.35}$	$\underline{42.60}_{3.13}$	$\underline{65.46}_{1.74}$	$\underline{79.34}_{1.23}$	$60.58_{16.76}$	$58.70_{2.17}$	$56.39_{28.20}$	$66.48_{4.02}$
UNIPELT (AP)	$86.26_{1.90}$	$\underline{82.77}_{1.09}$	$42.48_{3.38}$	$65.08_{1.65}$	$78.86_{1.45}$	$\underline{77.83}_{1.29}$	$\underline{59.46}_{3.71}$	$68.95_{2.14}$	$\underline{70.21}_{0.78}$
UNIPELT (APL)	$86.10_{1.28}$	$\mathbf{83.16}_{0.92}$	$\mathbf{43.83}_{4.73}$	$64.02_{2.99}$	$\mathbf{79.56}_{1.49}$	$\mathbf{78.54}_{1.95}$	$57.08_{3.87}$	$\mathbf{69.56}_{0.89}$	$\mathbf{70.23}_{0.55}$
[$K = 500$] Test Performance									
Fine-tuning	$\mathbf{85.67}_{0.97}$	$\underline{83.34}_{0.55}$	$36.47_{2.69}$	$59.64_{1.10}$	$77.30_{0.49}$	$\underline{84.96}_{1.19}$	$55.84_{0.85}$	$\underline{68.23}_{1.39}$	$68.93_{0.65}$
BitFit	$83.44_{0.63}$	$82.16_{0.37}$	$3.32_{2.59}$	$61.88_{2.75}$	$69.15_{9.91}$	$76.30_{0.36}$	$40.82_{3.30}$	$65.29_{3.66}$	$60.30_{1.91}$
Adapter	$84.54_{1.37}$	$82.53_{0.36}$	$38.65_{3.97}$	$59.35_{3.09}$	$77.39_{0.84}$	$83.52_{0.33}$	$50.04_{1.72}$	$68.12_{0.95}$	$68.02_{0.77}$
Prefix-tuning	$83.65_{0.69}$	$82.96_{1.63}$	$38.16_{2.25}$	$63.18_{2.70}$	$78.50_{1.12}$	$79.75_{1.49}$	$\mathbf{58.06}_{1.04}$	$54.34_{25.91}$	$67.32_{3.42}$
LoRA	$\underline{84.98}_{1.10}$	$82.53_{0.70}$	$\mathbf{39.86}_{2.71}$	$63.03_{2.57}$	$\mathbf{79.46}_{0.66}$	$65.05_{26.31}$	$\underline{56.54}_{2.05}$	$55.46_{27.74}$	$65.86_{4.18}$
UNIPELT (AP)	$84.84_{0.28}$	$83.25_{0.51}$	$\underline{39.84}_{5.01}$	$\underline{63.32}_{1.72}$	$78.36_{1.06}$	$84.53_{0.48}$	$56.08_{3.26}$	$68.14_{1.39}$	$\underline{69.79}_{1.02}$
UNIPELT (APL)	$84.91_{1.41}$	$\mathbf{83.56}_{0.59}$	$39.81_{2.55}$	$\mathbf{64.12}_{2.45}$	$\underline{79.28}_{0.63}$	$\mathbf{85.26}_{0.70}$	$54.07_{3.74}$	$\mathbf{68.87}_{0.41}$	$\mathbf{69.98}_{0.42}$
[$K = 1000$] Dev Performance									
Fine-tuning	$87.70_{0.89}$	$84.73_{0.61}$	$42.61_{2.62}$	$64.90_{2.01}$	$78.86_{2.00}$	$81.31_{1.39}$	$63.74_{1.59}$	$\mathbf{71.99}_{1.59}$	$71.98_{0.59}$
BitFit	$86.30_{1.36}$	$83.63_{0.18}$	$20.45_{16.56}$	$64.24_{1.55}$	$76.76_{0.84}$	$66.65_{0.87}$	$53.22_{1.73}$	$68.95_{2.32}$	$65.02_{2.12}$
Adapter	$87.06_{1.44}$	$84.79_{0.42}$	$43.48_{1.46}$	$65.62_{0.93}$	$79.88_{1.26}$	$80.88_{1.89}$	$59.56_{2.46}$	$70.52_{1.48}$	$71.47_{0.33}$
Prefix-tuning	$\underline{87.86}_{1.23}$	$83.48_{1.15}$	$44.04_{2.74}$	$\mathbf{68.08}_{0.81}$	$79.60_{1.61}$	$75.47_{2.92}$	$\underline{65.48}_{0.48}$	$68.94_{0.93}$	$71.62_{0.54}$
LoRA	$87.50_{1.01}$	$85.09_{1.02}$	$\mathbf{47.11}_{3.02}$	$\underline{67.20}_{0.78}$	$\mathbf{80.86}_{1.88}$	$76.33_{1.28}$	$62.86_{1.53}$	$\underline{71.48}_{1.45}$	$72.30_{0.52}$
UNIPELT (AP)	$87.32_{1.73}$	$\underline{85.52}_{0.63}$	$45.48_{3.52}$	$66.60_{0.99}$	$\underline{80.70}_{1.59}$	$\underline{82.96}_{1.47}$	$\mathbf{65.56}_{2.09}$	$70.58_{1.44}$	$\underline{73.09}_{0.46}$
UNIPELT (APL)	$\mathbf{88.02}_{1.28}$	$\mathbf{86.05}_{0.73}$	$\underline{45.70}_{2.47}$	$66.86_{1.32}$	$80.50_{1.76}$	$\mathbf{83.09}_{1.55}$	$64.60_{0.72}$	$70.64_{0.77}$	$\mathbf{73.18}_{0.27}$
[$K = 1000$] Test Performance									
Fine-tuning	$\underline{86.54}_{1.01}$	$84.87_{0.64}$	$43.26_{2.60}$	$62.31_{2.10}$	$79.03_{1.11}$	$86.39_{0.34}$	$61.95_{1.20}$	$\mathbf{71.09}_{0.77}$	$71.93_{0.37}$
BitFit	$83.99_{0.39}$	$83.95_{0.81}$	$22.44_{17.10}$	$62.89_{1.40}$	$77.43_{0.53}$	$79.04_{0.61}$	$52.87_{0.72}$	$69.50_{0.16}$	$66.51_{2.22}$
Adapter	$85.60_{0.63}$	$84.49_{0.60}$	$42.33_{1.98}$	$61.81_{1.57}$	$79.68_{0.23}$	$85.52_{0.29}$	$57.86_{2.44}$	$70.32_{0.71}$	$70.95_{0.55}$
Prefix-tuning	$85.09_{0.99}$	$83.66_{1.82}$	$44.07_{2.90}$	$\mathbf{66.71}_{2.72}$	$80.34_{0.70}$	$82.38_{1.25}$	$\mathbf{63.59}_{1.12}$	$68.58_{0.35}$	$71.81_{0.52}$
LoRA	$86.26_{1.22}$	$\underline{86.04}_{0.99}$	$\mathbf{45.50}_{1.11}$	$\underline{65.63}_{2.11}$	$\underline{81.00}_{0.98}$	$81.56_{1.97}$	$61.32_{1.65}$	$70.89_{0.81}$	$72.28_{0.69}$
UNIPELT (AP)	$86.17_{0.37}$	$85.86_{1.05}$	$44.33_{3.55}$	$64.91_{1.92}$	$80.65_{0.57}$	$\underline{86.82}_{0.23}$	$62.17_{0.99}$	$69.95_{0.90}$	$\underline{72.61}_{0.53}$
UNIPELT (APL)	$\mathbf{87.06}_{0.81}$	$\mathbf{86.65}_{1.10}$	$\underline{45.44}_{1.97}$	$65.49_{1.92}$	$\mathbf{81.22}_{0.51}$	$\mathbf{87.10}_{0.21}$	$\underline{62.49}_{0.94}$	$\underline{70.99}_{0.95}$	$\mathbf{73.31}_{0.52}$

Table 5: Results on the GLUE benchmark with $K = \{100, 500, 1000\}$ training samples. The evaluation metrics are Matthew's Correlation for CoLA, F1 for MRPC and QQP, Spearman's correlation for STS-B, and accuracy for the rest. For MNLI, we evaluate on the matched dataset. We report average performance on five random seeds with standard deviation as the subscript. **Best** and 2nd best methods under each setup are bold and underlined.