Adapters: A Unified Library for Parameter-Efficient and Modular Transfer Learning

Clifton Poth[*1,4], **Hannah Sterz**[*1], **Indraneil Paul**[1], **Sukannya Purkayastha**[1], **Leon Engländer**[1], **Timo Imhof**[1], **Ivan Vulić**[2], **Sebastian Ruder**[3], **Iryna Gurevych**[1], **Jonas Pfeiffer**[3]
[1]Ubiquitous Knowledge Processing Lab, Technical University of Darmstadt
[2]Language Technology Lab, University of Cambridge
[3]Google DeepMind [4]Cohere

Abstract

We introduce *Adapters*, an open-source library that unifies parameter-efficient and modular transfer learning in large language models. By integrating 10 diverse adapter methods into a unified interface, *Adapters* offers ease of use and flexible configuration. Our library allows researchers and practitioners to leverage adapter modularity through composition blocks, enabling the design of complex adapter setups. We demonstrate the library's efficacy by evaluating its performance against full fine-tuning on various NLP tasks. *Adapters* provides a powerful tool for addressing the challenges of conventional fine-tuning paradigms and promoting more efficient and modular transfer learning. The library is available via https://adapterhub.ml/adapters.

1 Introduction

The ever-increasing size of pretrained large language models (LLMs) (Brown et al., 2020; Chowdhery et al., 2022) has made the established transfer learning paradigm of fine-tuning all model parameters on a downstream task (Howard and Ruder, 2018; Devlin et al., 2019) extremely expensive. Moreover, the requirement of *parameter efficiency* at fine-tuning, while definitely paramount, is not the only shortcoming of the predominant LLM fine-tuning paradigm. It also suffers from other crucial issues such as negative interference, lack of positive transfer between tasks in multi-task learning (McCloskey and Cohen, 1989), catastrophic forgetting (French, 1999), and poor generalization.

Two closely related lines of research aimed at addressing this set of challenges have gained significant attention recently. First, *parameter-efficient fine-tuning* (Lialin et al., 2023; Sabry and Belz, 2023) focuses on the aspect of computational efficiency and feasibility by only fine-tuning a small number of parameters for downstream tasks. Second, *modular transfer learning* (Pfeiffer et al., 2023) focuses on the aspect of knowledge transfer by designing self-contained network modules which can be aggregated for better multi-task performance and generalization. In practice, these often represent two sides of the same coin. Methods that devise small components within a language model for fine-tuning on labeled task data, henceforth generally denoted as *adapters*, are both parameter-efficient *and* modular in nature.

The initial release of *AdapterHub* (Pfeiffer et al., 2020a) marks the first attempt to systematically make adapters accessible to researchers and practitioners in an easy-to-use framework. AdapterHub proposed a framework to easily integrate, train and use adapters for state-of-the-art Transformer models with minimal changes. It additionally established an open platform to share, explore and consume pre-trained adapter modules. While AdapterHub focused on bottleneck-style adapters (Houlsby et al., 2019) initially, the field of adapter methods has expanded substantially since (Li and Liang, 2021; Mahabadi et al., 2021; Hu et al., 2022; He et al., 2022; Liu et al., 2022, *among others*).

With increasing interest in adapter methods, new tools and libraries have been developed. *OpenDelta* (Hu et al., 2023a), HuggingFace's *PEFT* (Mangrulkar et al., 2022) and *LLM-Adapters* (Hu et al., 2023b) are recent examples of libraries which attempt to unify adapter methods in a single code base and extend their applicability to new model architectures. However, these works exclusively focus on the parameter-efficiency aspect of adapters, neglecting the modularity side of these methods.

Contributions. Based on the initial version of AdapterHub, we, therefore, propose *Adapters*, a new library aimed at *unifying parameter-efficient and modular transfer learning*. Compared to the first AdapterHub iteration and concurrent libraries, our main contributions can be summarized as fol-

[*]Authors contributed equally.

Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing: System Demonstrations, pages 149–160
December 6-10, 2023

lows: **1)** We propose a self-contained library that integrates 10 diverse adapter methods into a unified interface for easy usage and flexible configuration; **2)** we develop a simple way of leveraging the modularity of adapters by designing *composition blocks* that allow flexibly defining complex adapter setups; **3)** we integrate all methods into 20 Transformer-based models spanning NLP, vision, and multi-modal applications; **4)** we evaluate the performance of our adapter implementations against full fine-tuning on a diverse set of tasks.

2 Background

We use the term *adapter* in a more general sense to refer to a broad family of transfer learning methods that share the two defining properties: parameter efficiency and modularity. For a detailed overview of different adapter architectures, we refer the reader to the recent survey by Pfeiffer et al. (2023).

2.1 Parameter Efficiency

Let the parameters of a language model be composed of a set of pre-trained parameters Θ (frozen) and a set of parameters Φ (where Φ can either be newly introduced or $\Phi \subset \Theta$). During fine-tuning, adapter methods optimize only Φ according to a loss function L on a dataset D:

$$\Phi^* \leftarrow \arg\min_{\Phi} L(D; \{\Theta, \Phi\})$$

Different adapter methods insert parameters Φ at different locations of a pre-trained large model. Bottleneck adapters (Rebuffi et al., 2017; Houlsby et al., 2019), as one of the early methods, introduce bottleneck feed-forward layers in each layer of a Transformer model. Subsequent designs have adapted a Transformer model's self-attentions (Li and Liang, 2021), bias terms (Ben Zaken et al., 2022), input prompts (Lester et al., 2021) or embeddings (Pfeiffer et al., 2021b). Complementary lines of work have focused on optimizing the parameter efficiency (Mahabadi et al., 2021; Liu et al., 2022) and runtime efficiency (Hu et al., 2022; Lei et al., 2023) of adapters or have attempted to unify multiple components in a single framework (He et al., 2022; Mao et al., 2022).

2.2 Modularity

A modular deep learning model is composed of modules that each capture a specific functionality of the full model, such as task or language capacities. Pfeiffer et al. (2023) propose a taxonomy of modular deep learning methods covering the dimensions of computation function, routing, aggregation, and training.

Routing and aggregation are of special interest here as they coordinate the composition of multiple adapter modules, a key functionality enabled by modularity. Exemplary existing work includes using stochastic routing through adapters (Wang et al., 2022), adapter parameter averaging (Friedman et al., 2021), sequential function aggregation of adapter modules (Pfeiffer et al., 2022b) as well as weighted (Wang et al., 2021) and attention-based (Pfeiffer et al., 2021a) output aggregation.

Finally, along the training dimension, the modularity of adapters allows using pre-trained adapter modules as initialization for further fine-tuning (Poth et al., 2021; Vu et al., 2022).

	AdapterHub v1	**Adapters**
Design	Fork of Transformers	Self-contained add-on library
Adapter methods	2	10
Complex configurations	✗	✓
Composition blocks	✗[1]	✓ (6)
Model architectures	3	20
AdapterHub.ml / HF Hub integration	✓ / ✗	✓ / ✓

Table 1: Feature comparison between the initial *AdapterHub* release (Pfeiffer et al., 2020a) and the proposed *Adapters* library.

3 The *Adapters* Library

Adapters builds on many design decisions established in the initial *AdapterHub* release (Pfeiffer et al., 2020a), but offers substantial extensions both 'horizontally' (e.g., extending the support to many more pretrained neural architectures, extending the coverage of adapter architectures) and 'vertically' (e.g., adding new composition and processing capabilities). Table 1 gives an overview of the differences between the initial *AdapterHub* and *Adapters*. The core features adopted from the initial release, facilitating its ease of use and wider adoption by researchers and practitioners, include: **1)** Tight integration into the widely used HuggingFace Transformers (Wolf et al., 2020) library; **2)** adaptation of pre-existing Transformers fine-tuning scripts with minimal changes; **3)** single-line saving and loading of adapter modules from a shared community hub.

[1] V1 already supported stacking and fusing adapter, however without flexibly composable blocks.

```python
import adapters
from transformers import AutoModel

model = AutoModel.from_pretrained("..")
adapters.init(model)
model.add_adapter("a", config="seq_bn")
model.add_adapter("b", config="seq_bn")
model.train_adapter(Parallel("a", "b"))
```

Listing 1: Example of adding adapters to an existing Transformers model. After model instantiation, `init()` introduces adapter-specific functionality. Two bottleneck adapters are added via `add_adapter()` and activated for parallel training.

Method name	Default config
Bottleneck adapter (Houlsby et al., 2019)	`[double_]seq_bn`
Invertible adapter (Pfeiffer et al., 2020b)	`seq_bn_inv`
Prompt tuning (Lester et al., 2021)	`prompt_tuning`
Prefix tuning (Li and Liang, 2021)	`prefix_tuning`
Compacter (Mahabadi et al., 2021)	`compacter`
LoRA (Hu et al., 2022)	`lora`
$(IA)^3$ (Liu et al., 2022)	`ia3`
Parallel adapter (He et al., 2022)	`par_bn`
Mix-and-Match adapter (He et al., 2022)	`mam`
UniPELT (Mao et al., 2022)	`unipelt`

Table 2: Overview of adapter methods supported in the *Adapters* library at the time of submission (Aug 2023).

3.1 Transformers Integration

Unlike the initial AdapterHub, *Adapters* is designed as a standalone package that acts as an add-on to the Transformers library. *Adapters* provides adapter implementations and management methods that can be injected into pre-trained Transformer checkpoints without modifying the original model code directly. We provide two approaches to this end: (i) by attaching to existing models and (ii) by providing our own, specialized model classes.

Attaching to Models. The `init()` method provides a straightforward solution for making adapters accessible to pre-existing model classes post-hoc. A model checkpoint for one of the supported architectures (cf. § 3.5) can be instantiated via any of the model classes provided by the Transformers library. An example of this approach is given in Listing 1. All adapter-related functionality is injected post-instantiation by passing the model instance to the `init()` method. Afterwards, all methods provided by *Adapters* are easily invokable from the same model instance.

AdapterModel Classes. As an alternative to post-hoc initialization, *Adapters* provides a set of built-in model classes optimized for working with adapters. Following HuggingFace's naming conventions, these classes are named `XXXAdapterModel`, where `XXX` is the name of the model architecture. Compared to models with attached adapters, these classes especially provide more flexibility with regard to prediction heads. Each model instance can have multiple named prediction heads targeted towards different tasks loaded simultaneously. These prediction heads can be associated with adapter modules by sharing a common name.

Providing this functionality is crucial for enabling features such as quickly switching between adapters targeted toward different tasks at runtime. It is also essential for creating composed adapter setups such as parallel inference on multiple tasks (cf. § 3.4). We, therefore, provide automatic conversion from HuggingFace's model classes - typically paired with a single, fixed prediction head - to our newly introduced classes featuring flexible prediction heads.

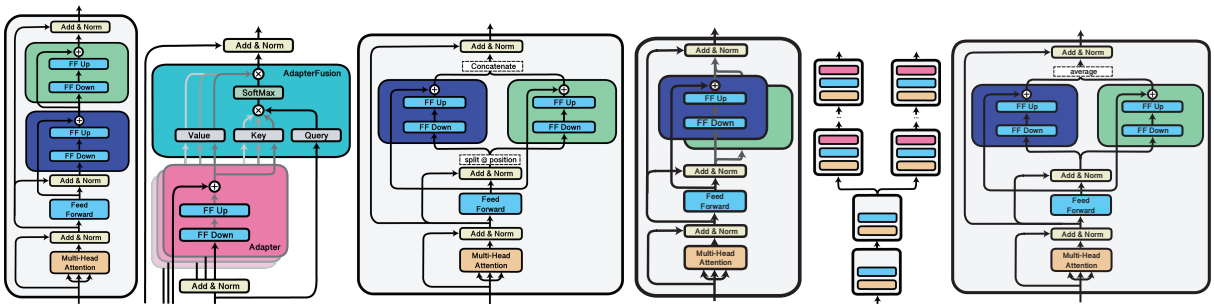
3.2 Unified Adapter Interface

Adapters defines a common interface of methods covering the full life cycle of working with adapters. This includes methods for adding, activating, saving, releasing, loading, aggregating, and deleting adapter modules. When adding a new adapter to a model (e.g., via `add_adapter()`), it is given a unique identifier string. All adapter-related methods then solely use this string to identify the adapter module an operation should be performed on. Thus, the adapter interface at the model level can be agnostic to specific adapter methods.

This interface, as well as adapter implementations at the module level, are integrated into model classes via Python mixins and dynamically modifying Python classes at runtime to keep changes to the existing Transformers code base minimal.

3.3 Adapter Methods

Each adapter method is defined by a configuration object or string which allow flexible customization of various properties of an adapter module, including placement, capacity, residual connections, initialization etc. We distinguish between single methods consisting of one type of adapter module and complex methods consisting of multiple different adapter module types. Table 2 gives an overview of all methods currently integrated into *Adapters*, along with their configuration strings.



(a) Stack (b) Fuse (c) Split (d) BatchSplit (e) Parallel (f) Average

Figure 1: Overview of adapter composition blocks supported in the *Adapters* library at the time of writing this paper.

3.3.1 Single Methods

Adapters supports single adapter methods that introduce parameters in new feed-forward modules such as bottleneck adapters (Houlsby et al., 2019), introduce prompts at different locations such as prefix tuning (Li and Liang, 2021), reparameterize existing modules such as LoRA (Hu et al., 2022) or re-scale their output representations such as $(IA)^3$ (Liu et al., 2022). Detailed descriptions for all currently implemented single methods are given in Appendix A, see also Table 2 again.

3.3.2 Complex Methods

While different efficient fine-tuning methods and configurations have often been proposed as standalone, combining them for joint training has proven to be beneficial (He et al., 2022; Mao et al., 2022). To make this process easier, *Adapters* provides the possibility to group multiple configuration instances using the `ConfigUnion` class. This flexible mechanism allows easy integration of multiple complex methods proposed in the literature (as the two examples outlined below), as well as the construction of other, new complex configurations currently not available nor benchmarked in the literature (Zhou et al., 2023).

Mix-and-Match Adapters (He et al., 2022) was proposed as a combination of Prefix-Tuning and parallel bottleneck adapters. Using `ConfigUnion`, this method can be defined as:

```
config = ConfigUnion(
  PrefixTuningConfig(bottleneck_size=800),
  ParallelConfig(),
)
model.add_adapter("name", config=config)
```

UniPELT (Mao et al., 2022) combines LoRA, Prefix Tuning, and bottleneck adapters in a single unified setup. It additionally introduces a gating mechanism that controls the activation of the different adapter modules. $\mathcal{G}_m \leftarrow \sigma(W_{\mathcal{G}_m} \cdot x)$.

3.4 Adapter Composition

While the modularity and composability aspects of adapters have seen increasing interest in research, existing open-source libraries (Mangrulkar et al., 2022; Hu et al., 2023a) have largely overlooked these aspects. *Adapters* makes adapter compositions a central and accessible part of working with adapters by enabling the definition of complex, composed adapter setups. We define a set of simple *composition blocks* that each capture a specific method of aggregating the functionality of multiple adapters. Each composition block class takes a sequence of adapter identifiers plus optional configuration as arguments. The defined adapter setup is then parsed at runtime by *Adapters* to allow for dynamic switching between adapters per forward pass. Fig. 1 shows schematic illustrations of all composition blocks supported by *Adapters*. Listing 2 shows examples of how the same composition blocks are defined in code:

```
Stack("a", "b", "c")
Fuse("d", "e", "f")
Split("g", "h", splits=[64, 64])
BatchSplit("i", "j", batch_sizes=[2, 4])
Parallel("k", "l", "m")
Average("n", "o", weights=[0.3, 0.7])
Stack("p", Parallel("q", "r"))
```

Listing 2: Code examples of composition blocks supported by *Adapters*. Strings represent adapter IDs.

In what follows, we present each supported composition in more detail.

Stack. The Stack block allows stacking multiple adapters sequentially within a Transformer layer. This type of composition is, e.g., used in the MAD-X framework for cross-lingual transfer (Pfeiffer et al., 2020b), where language and task adapters are stacked. In Listing 2, the input is first passed through *a*, the output of *a* is then inputted to *b*, and the output of *b* is finally inputted to *c*.

Fuse. The Fuse block can be used to activate an AdapterFusion layer (Pfeiffer et al., 2021a). AdapterFusion is a non-destructive way to combine the knowledge of multiple pre-trained adapters on a new downstream task. In Listing 2, we activate the adapters *d*, *e*, and *f* as well as the fusion layer that combines the outputs of all three.[2]

Split. The Split block can be used to split an input sequence between multiple adapters. This e.g. enables splitting multimodal input sequences to modality-specific adapters (Pfeiffer et al., 2022a). In Listing 2, we split each input sequence between adapters *g* and *h*. All tokens with indices 0 - 63 are forwarded through *g* while the next 64 tokens beginning at index 64 are forwarded through *h*.

BatchSplit. The BatchSplit block splits inputs along the batch size dimension between several adapters. That is, different adapters receive different sub-batches of the full input batch. In Listing 2, we split the input batch between adapters *i*, and *j*. Adapter *i* receives two sequences and *j* receives four sequences. The sum of all specified sub-batches has to match the batch size of the input.

Parallel. This block can be used to enable independent parallel training and inference on different adapters, where each adapter has its own prediction head. The implementation automatically replicates all inputs at the first occurrence of parallel adapter modules, sharing the inputs in all lower layers without parallel modules. This mechanism was first used in Rücklé et al. (2021). In Listing 2, we forward all inputs via adapters *k*, *l*, and *m* in parallel.

Average. Following approaches of ensembling full models at inference time for better generalization, recent work has explored methods of averaging pre-trained adapters. This includes averaging adapter output representations (Wang et al., 2021) as well as averaging adapter parameters (Friedman et al., 2021; Wang et al., 2022; Chronopoulou et al., 2023). *Adapters* provides built-in support for both types of inference time-averaging methods. **Output averaging** allows to dynamically aggregate the output representations of multiple adapters in a model forward pass via weighted averaging. This is realized via the Average composition block. In Listing 2, two adapters are averaged with the weights 0.3 for *n* and 0.7 for *o*. **Parameter averaging** enables creating a new adapter via weighted averaging of the parameters of multiple pre-trained adapters. As this process is typically not done dynamically at runtime, *Adapters* provides `average_adapter()` as a dedicated method. Compared to output averaging, parameter averaging of adapters has the advantage of not inducing any additional inference time relative to using a single adapter.

Nesting. Finally, it is possible to nest composition blocks within other composition blocks to create deeper and more complex compositions. *Adapters* defines a set of allowed nestings to restrict the users to setups that are sensible. As an example, we nest a Parallel block within a Stack block in Listing 2.

3.5 Supported Models

At the time of release, *Adapters* has built-in support for 20 widely adopted model architectures included in the Transformers library. This covers text encoder models such as BERT (Devlin et al., 2019) and DeBERTa (He et al., 2021), text decoder models such as GPT-2 (Radford et al., 2019), sequence-to-sequence models such as BART (Lewis et al., 2020) and T5 (Raffel et al., 2020), vision encoder models such as ViT (Dosovitskiy et al., 2021), as well as multimodal models such as CLIP (Radford et al., 2021).[3]

While adapter-related implementations mostly can be shared across all supported models, correctly integrating them into each model implementation requires manual effort. While it is difficult to standardize this process due to differences between model architectures, we provide clear guidelines for integrating new models in the form of shared interfaces and step-by-step documentation[4].

3.6 AdapterHub Ecosystem

Adapters is integrated into the extensive existing open-source ecosystem introduced by AdapterHub (Pfeiffer et al., 2020a). Most prominently, this includes AdapterHub.ml as a platform to share and discover pre-trained adapter modules. *Adapters* further broadens the possibilities for sharing adapters by integrating with HuggingFace's Model Hub, which has emerged as one of the primary platforms for open-sourcing model checkpoints. The new Hub integration comes with programmatic methods of discovering and publishing pre-trained adapter

[2]Note that this requires a fusion layer to be added beforehand via `add_adapter_fusion()`.

[3]An up-to-date full list of supported models can be found at https://docs.adapterhub.ml/model_overview.html.

[4]https://docs.adapterhub.ml/contributing/adding_adapters_to_a_model.html

Method	Sequence Classification							Regression	Mult. Choice	Extract. QA	Tagging	Avg.
	CoLA Dev MCC	MNLI Dev Acc.	MRPC Dev F1	QNLI Dev Acc.	QQP Dev F1	RTE Dev Acc.	SST-2 Dev Acc.	STS-B Dev PCC	Cosmos QA Dev Acc.	SQuAD v2 Dev F1	CoNLL-2003 Test F1	
double_seq_bn	63.58 (±19.19)	87.61 (±26.41)	**93.31** (±4.52)	92.84 (±17.17)	91.58 (±36.83)	**80.87** (±11.09)	94.73 (±17.51)	90.85 (±27.16)	70.99 (±16.87)	**84.89** (±5.52)	91.92 (±17.65)	**85.74** (±18.17)
seq_bn	**71.22** (±23.40)	87.5 (±20.39)	92.91 (±4.54)	**93.15** (±15.83)	89.69 (±21.31)	79.42 (±9.81)	95.18 (±13.26)	89.44 (±20.33)	69.68 (±16.44)	82.88 (±**1.04**)	**92.02** (±11.48)	**85.74** (±14.34)
par_bn	63.95 (±23.72)	87.44 (±21.66)	93.24 (±4.65)	93.04 (±17.26)	88.32 (±33.14)	77.98 (±10.95)	94.95 (±16.97)	90.33 (±**5.64**)	**80.10** (±18.47)	82.56 (±6.70)	91.95 (±27.60)	85.81 (±16.98)
compacter	55.52 (±**13.67**)	86.10 (±**1.99**)	90.43 (±**3.58**)	92.42 (±**2.68**)	86.68 (±**2.14**)	68.59 (±**4.91**)	94.15 (±**0.81**)	90.06 (±23.27)	67.91 (±10.42)	79.20 (±8.87)	91.27 (±8.58)	82.03 (±**7.36**)
prefix_tuning	61.62 (±4.93)	86.98 (±18.91)	91.06 (±4.09)	92.46 (±9.55)	87.07 (±15.58)	71.12 (±6.06)	**95.18** (±0.54)	90.13 (±29.23)	66.13 (±**3.44**)	78.16 (±2.41)	91.46 (±**2.44**)	82.85 (±8.79)
lora	63.99 (±20.64)	87.59 (±4.29)	92.60 (±4.39)	93.11 (±3.77)	88.48 (±2.57)	80.26 (±9.28)	94.99 (±8.48)	90.72 (±19.31)	70.63 (±8.65)	82.46 (±8.86)	91.85 (±21.68)	85.15 (±10.17)
ia3	63.03 (±21.39)	86.19 (±5.08)	92.32 (±3.94)	91.88 (±3.73)	86.41 (±13.46)	76.89 (±7.17)	94.42 (±2.13)	90.65 (±29.16)	66.85 (±9.69)	78.52 (±10.11)	91.56 (±21.94)	83.52 (±11.62)
Full Fine-tuning	63.66	**87.63**	90.20	92.81	**91.92**	78.77	94.81	**91.20**	68.87	82.91	91.33	84.91

Table 3: Best performance (± std. dev. across all hyper-parameters) of various supported single adapter methods (cf. §3.3.1) applied to `roberta-base`, benchmarked against full fine-tuning. The best results and lowest std. dev. per task are highlighted in **bold**.

modules, in addition to the previously available methods for downloading and saving adapters.

At the time of writing, users of *Adapters* have access to over 700 pre-trained adapters.

4 Adapter Evaluation

In addition to the ease of use aforementioned, we show that the adapter methods offered by our library are performant across a range of settings. To this end, we conduct evaluations on the single adapter implementations made available by *Adapters* (see §3.3.1). We demonstrate the effectiveness of these methods against full fine-tuning on a variety of task types: extractive question answering (Rajpurkar et al., 2018), multiple choice classification (Huang et al., 2019), sequence tagging (Tjong Kim Sang, 2002), sequence to sequence summarization (Narayan et al., 2018), sequence classification and sequence regression (Wang et al., 2019). To make our evaluations reflective of user experience, we conduct them using the two most commonly used base model variants on AdapterHub: *roberta-base* (Liu et al., 2019) and *bart-base* (Lewis et al., 2020).

Setup. We conduct a grid search over a range of common training hyper-parameters, varying learning rates between $\{10^{-5}, 10^{-4}, 5 \cdot 10^{-4}, 10^{-4}, 10^{-3}\}$ and the number of epochs between $\{5, 10, 20, 30\}$. We also augment the grid with a number of adapter-specific hyper-parameters. These, along with the minimum and maximum trainable parameters added across the configurations, are detailed in Table A. The highest attained performance (and the standard deviation of results across the grid) for the two chosen base models are outlined in Tables 3 and 4, respectively.

Results. The obvious takeaway from our evaluations is that all adapter implementations offered by our framework are competitive with full model fine-tuning, across all task classes. Approaches that offer more tunable hyper-parameters (and thus allow for easy scaling) such as Bottleneck adapters, LoRA, and Prefix Tuning predictably have the highest topline performance, often surpassing full fine-tuning. However, extremely parameter-frugal methods like $(IA)^3$, which add $< 0.005\%$ of the parameters of the base model, also perform commendably and only fall short by a small fraction. Finally, the Compacter is the least volatile among the single methods, obtaining the lowest standard deviation between runs on the majority of tasks.

5 Conclusion

We have presented *Adapters*, a novel library for research and application of adapters. Unlike comparable solutions, *Adapters* equally focuses on the parameter-efficiency and modularity side of adapters. Our library implements a diverse set of adapter methods under a unified interface which allows flexible configuration and mixing of different approaches. We proposed a simple building block system for leveraging the modularity of adapters to build complex adapter setups. *Adapters* tightly integrates into the HuggingFace and AdapterHub ecosystems and its adapter implementations show performances competitive to full fine-tuning.

As research on adapters and LLMs continues to advance rapidly, our library will evolve as well. Its extensibility makes *Adapters* well prepared for the integration of new adapter methods and model architectures, both from us and the community.

Acknowledgements

This work has been funded by Huawei Technologies (Ireland) Co., Ltd. and German Research Foundation (DFG) as part of the Research Training Group KRITIS No. GRK 2222. We thank Martin Hentschel for his help building the demo web app for our library. We thank Francesco Piccinno and Srini Narayanan for their helpful comments and suggestions during the initial drafts of this paper. We also thank the many contributors and users of the *AdapterHub* open-source framework.

References

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)*, pages 1–9, Dublin, Ireland.

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. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*.

Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, Parker Schuh, Kensen Shi, Sasha Tsvyashchenko, Joshua Maynez, Abhishek Rao, Parker Barnes, Yi Tay, Noam Shazeer, Vinodkumar Prabhakaran, Emily Reif, Nan Du, Ben Hutchinson, Reiner Pope, James Bradbury, Jacob Austin, Michael Isard, Guy Gur-Ari, Pengcheng Yin, Toju Duke, Anselm Levskaya, Sanjay Ghemawat, Sunipa Dev, Henryk Michalewski, Xavier Garcia, Vedant Misra, Kevin Robinson, Liam Fedus, Denny Zhou, Daphne Ippolito, David Luan, Hyeontaek Lim, Barret Zoph, Alexander Spiridonov, Ryan Sepassi, David Dohan, Shivani Agrawal, Mark Omernick, Andrew M. Dai, Thanumalayan Sankaranarayana Pillai, Marie Pellat, Aitor Lewkowycz, Erica Moreira, Rewon Child, Oleksandr Polozov, Katherine Lee, Zongwei Zhou, Xuezhi Wang, Brennan Saeta, Mark Diaz, Orhan Firat, Michele Catasta, Jason Wei, Kathy Meier-Hellstern, Douglas Eck, Jeff Dean, Slav Petrov, and Noah Fiedel. 2022. PaLM: Scaling Language Modeling with Pathways. *ArXiv preprint*.

Alexandra Chronopoulou, Matthew Peters, Alexander Fraser, and Jesse Dodge. 2023. AdapterSoup: Weight averaging to improve generalization of pretrained language models. In *Findings of the Association for Computational Linguistics: EACL 2023*, pages 2054–2063, Dubrovnik, Croatia.

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.

Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. 2021. An image is worth 16x16 words: Transformers for image recognition at scale. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*.

Robert M. French. 1999. Catastrophic forgetting in connectionist networks. *Trends in Cognitive Sciences*, (4):128–135.

Dan Friedman, Ben Dodge, and Danqi Chen. 2021. Single-dataset experts for multi-dataset question answering. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 6128–6137, Online and Punta Cana, Dominican Republic.

Junxian He, Chunting Zhou, Xuezhe Ma, Taylor Berg-Kirkpatrick, and Graham Neubig. 2022. Towards a unified view of parameter-efficient transfer learning. In *The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, April 25-29, 2022*.

Pengcheng He, Xiaodong Liu, Jianfeng Gao, and Weizhu Chen. 2021. Deberta: decoding-enhanced bert with disentangled attention. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*.

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, pages 2790–2799.

Jeremy Howard and Sebastian Ruder. 2018. Universal language model fine-tuning for text classification. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 328–339, Melbourne, Australia.

Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2022. Lora: Low-rank adaptation of large language models. In *The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, April 25-29, 2022*.

Shengding Hu, Ning Ding, Weilin Zhao, Xingtai Lv, Zhen Zhang, Zhiyuan Liu, and Maosong Sun. 2023a. OpenDelta: A plug-and-play library for parameter-efficient adaptation of pre-trained models. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 3: System Demonstrations)*, pages 274–281, Toronto, Canada.

Zhiqiang Hu, Yihuai Lan, Lei Wang, Wanyu Xu, Ee-Peng Lim, Roy Ka-Wei Lee, Lidong Bing, and Soujanya Poria. 2023b. Llm-adapters: An adapter family for parameter-efficient fine-tuning of large language models. *ArXiv preprint*.

Lifu Huang, Ronan Le Bras, Chandra Bhagavatula, and Yejin Choi. 2019. Cosmos QA: Machine reading comprehension with contextual commonsense reasoning. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 2391–2401, Hong Kong, China.

Tao Lei, Junwen Bai, Siddhartha Brahma, Joshua Ainslie, Kenton Lee, Yanqi Zhou, Nan Du, Vincent Y. Zhao, Yuexin Wu, Bo Li, Yu Zhang, and Ming-Wei Chang. 2023. Conditional adapters: Parameter-efficient transfer learning with fast inference. *ArXiv preprint*.

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*, pages 3045–3059, Online and Punta Cana, Dominican Republic.

Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Veselin Stoyanov, and Luke Zettlemoyer. 2020. BART: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 7871–7880, Online.

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.

Vladislav Lialin, Vijeta Deshpande, and Anna Rumshisky. 2023. Scaling Down to Scale Up: A Guide to Parameter-Efficient Fine-Tuning. *ArXiv preprint*.

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. In *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, Nov 28 - Dec 9, 2022, virtual and New Orleans, LA, USA*, pages 1950–1965.

Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. Ro{bert}a: A robustly optimized {bert} pretraining approach. *ArXiv preprint*.

Rabeeh Karimi Mahabadi, James Henderson, and Sebastian Ruder. 2021. Compacter: Efficient low-rank hypercomplex adapter layers. In *Advances in Neural Information Processing Systems 34: Annual Conference on Neural Information Processing Systems 2021, NeurIPS 2021, December 6-14, 2021, virtual*, pages 1022–1035.

Sourab Mangrulkar, Sylvain Gugger, Lysandre Debut, Younes Belkada, and Sayak Paul. 2022. Peft: State-of-the-art parameter-efficient fine-tuning methods. `https://github.com/huggingface/peft`.

Yuning Mao, Lambert Mathias, Rui Hou, Amjad Almahairi, Hao Ma, Jiawei Han, Scott Yih, and Madian Khabsa. 2022. UniPELT: A unified framework for parameter-efficient language model tuning. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 6253–6264, Dublin, Ireland.

Michael McCloskey and Neal J Cohen. 1989. Catastrophic interference in connectionist networks: The sequential learning problem. In *Psychology of learning and motivation*, volume 24, pages 109–165.

Shashi Narayan, Shay B. Cohen, and Mirella Lapata. 2018. Don't give me the details, just the summary! topic-aware convolutional neural networks for extreme summarization. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 1797–1807, Brussels, Belgium.

Jonas Pfeiffer, Gregor Geigle, Aishwarya Kamath, Jan-Martin Steitz, Stefan Roth, Ivan Vulić, and Iryna Gurevych. 2022a. xGQA: Cross-lingual visual question answering. In *Findings of the Association for Computational Linguistics: ACL 2022*, pages 2497–2511, Dublin, Ireland.

Jonas Pfeiffer, Naman Goyal, Xi Lin, Xian Li, James Cross, Sebastian Riedel, and Mikel Artetxe. 2022b. Lifting the curse of multilinguality by pre-training modular transformers. In *Proceedings of the 2022 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 3479–3495, Seattle, United States.

Jonas Pfeiffer, Aishwarya Kamath, Andreas Rücklé, Kyunghyun Cho, and Iryna Gurevych. 2021a. 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.

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: System Demonstrations*, pages 46–54, Online.

Jonas Pfeiffer, Sebastian Ruder, Ivan Vuli'c, and Edoardo M. Ponti. 2023. Modular Deep Learning. *ArXiv preprint*.

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.

Jonas Pfeiffer, Ivan Vulić, Iryna Gurevych, and Sebastian Ruder. 2021b. UNKs everywhere: Adapting multilingual language models to new scripts. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 10186–10203, Online and Punta Cana, Dominican Republic.

Clifton Poth, Jonas Pfeiffer, Andreas Rücklé, and Iryna Gurevych. 2021. What to pre-train on? Efficient intermediate task selection. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 10585–10605, Online and Punta Cana, Dominican Republic.

Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, Gretchen Krueger, and Ilya Sutskever. 2021. Learning transferable visual models from natural language supervision. In *Proceedings of the 38th International Conference on Machine Learning, ICML 2021, 18-24 July 2021, Virtual Event*, Proceedings of Machine Learning Research, pages 8748–8763.

Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2019. Language models are unsupervised multitask learners. *Technical report*.

Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. 2020. Exploring the limits of transfer learning with a unified text-to-text transformer. *J. Mach. Learn. Res.*, pages 140:1–140:67.

Pranav Rajpurkar, Robin Jia, and Percy Liang. 2018. Know what you don't know: Unanswerable questions for SQuAD. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 784–789, Melbourne, Australia.

Sylvestre-Alvise Rebuffi, Hakan Bilen, and Andrea Vedaldi. 2017. Learning multiple visual domains with residual adapters. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, pages 506–516.

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*, pages 7930–7946, Online and Punta Cana, Dominican Republic.

Mohammed Sabry and Anya Belz. 2023. Peft-ref: A modular reference architecture and typology for parameter-efficient finetuning techniques. *ArXiv preprint*.

Erik F. Tjong Kim Sang. 2002. Introduction to the CoNLL-2002 shared task: Language-independent named entity recognition. In *COLING-02: The 6th Conference on Natural Language Learning 2002 (CoNLL-2002)*.

Tu Vu, Brian Lester, Noah Constant, Rami Al-Rfou', and Daniel Cer. 2022. SPoT: Better frozen model adaptation through soft prompt transfer. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 5039–5059, Dublin, Ireland.

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 *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*.

Xinyi Wang, Yulia Tsvetkov, Sebastian Ruder, and Graham Neubig. 2021. Efficient test time adapter ensembling for low-resource language varieties. In *Findings of the Association for Computational Linguistics: EMNLP 2021*, pages 730–737, Punta Cana, Dominican Republic.

Yaqing Wang, Sahaj Agarwal, Subhabrata Mukherjee, Xiaodong Liu, Jing Gao, Ahmed Hassan Awadallah, and Jianfeng Gao. 2022. AdaMix: Mixture-of-adaptations for parameter-efficient model tuning. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pages 5744–5760, Abu Dhabi, United Arab Emirates.

Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Remi 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 Rush. 2020. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 38–45, Online.

Han Zhou, Xingchen Wan, Ivan Vulić, and Anna Korhonen. 2023. AutoPEFT: Automatic configuration search for parameter-efficient fine-tuning. *ArXiv preprint*.

A Description of Adapter Methods

Bottleneck Adapters introduce bottleneck modules in each layer of a Transformer model. Generally, these adapter modules consist of a down-projection matrix W_{down} that projects into a lower dimension $d_{bottleneck}$, a non-linearity f, an up-projection W_{up} that projects back into the original hidden layer dimension and a residual connection r, i.e.: $h \leftarrow W_{up} \cdot f(W_{down} \cdot h) + r$.

Depending on the specific configuration, these layers can be introduced at different locations and in **sequential** or **parallel** order relative to the adapted Transformer layer. *Adapters* provides predefined configurations for the sequential configurations of Houlsby et al. (2019) and Pfeiffer et al. (2021a) as well as the parallel configuration of He et al. (2022).

Invertible Adapters were proposed as part of MAD-X (Pfeiffer et al., 2020b) to learn language-specific transformations. Embedding outputs are passed through an invertible adapter module in the forward direction before entering the first Transformer layer and in the inverse direction after leaving the last Transformer layer.

Prompt Tuning (Lester et al., 2021) is an approach to condition language models on a task-specific soft prompt. While hard prompts consist of fixed textual descriptions prepended to the model's input (Brown et al., 2020), these soft prompts are continuously optimized towards the target task via gradient descent. Prompt tokens are prepended to the embedded input sequence.

Prefix Tuning (Li and Liang, 2021) is an extension of prompt tuning which prepends trainable prefix vectors P^K and P^V to the keys and values of the multi-head attention block inputs. The prefix vectors have a configurable length and are not optimized directly but reparameterized via a bottleneck MLP in the built-in default configuration, following the original implementation.

Compacter (Mahabadi et al., 2021) exchanges the linear down- and up-projection of a bottleneck adapter for a PHM layer[5]. This PHM layer constructs its weight matrix from two smaller matrices, which reduces the number of parameters needed for the adapters. These matrices can be factorized and shared between all adapter layers. *Adapters* provides pre-defined configurations for the Compacter and Compacter++ variants.

LoRA (Hu et al., 2022) injects trainable low-rank decomposition matrices into the layers of a pre-trained model. For any model layer expressed as a matrix multiplication of the form $h = W_0 x$, it performs a reparameterization such that: $h = W_0 x + \frac{\alpha}{r} BAx$. Here, $A \in \mathbb{R}^{r \times k}$ and $B \in \mathbb{R}^{d \times r}$ are the decomposition matrices and r is the low-dimensional rank of the decomposition. With *Adapters*, LoRA modules can be configured to be placed into the self-attention, intermediate, or output components of a Transformer layer. Following Hu et al. (2022), *Adapters* provides a built-in method of merging LoRA modules with the original pre-trained weights of a model for inference without additional latency.

(IA)³ (Liu et al., 2022) introduces trainable vectors l_W into different components of a Transformer model, which perform element-wise rescaling of inner model activations. For any model layer expressed as a matrix multiplication of the form $h = Wx$, it therefore performs an element-wise multiplication with l_W, such that: $h = l_W \odot Wx$.

[5] Parametrized hypercomplex multiplication layer.

Method	Sequence Classification					Regression	Extract. QA	Seq2Seq	Avg.
	CoLA Dev MCC	MRPC Dev F1	QNLI Dev Acc.	RTE Dev Acc.	SST-2 Dev Acc.	STS-B Dev PCC	SQuAD v2 Dev F1	XSum Rouge 2	
double_seq_bn	61.80 (±21.47)	91.62 (±3.73)	92.18 (±15.35)	73.65 (±8.26)	94.04 (±13.89)	89.89 (±23.18)	79.37 (±12.17)	17.49 (±1.51)	75.00 (±10.86)
seq_bn	53.29 (±18.82)	91.16 (±3.59)	92.11 (±6.29)	75.45 (±6.84)	93.46 (±**1.09**)	89.44 (±20.38)	79.47 (±10.41)	17.76 (±1.60)	74.02 (±7.55)
par_bn	55.58 (±19.70)	90.75 (±3.46)	92.12 (±14.42)	75.09 (±7.91)	94.08 (±14.39)	89.58 (±**10.86**)	79.63 (±**8.19**)	**18.26** (± 1.19)	74.39 (±10.01)
compacter	44.81 (±12.06)	87.44 (±4.30)	91.45 (±**3.14**)	69.68 (±**3.85**)	93.35 (±1.18)	87.62 (±14.09)	74.33 (±9.81)	13.55 (±5.29)	70.28 (±**6.72**)
prefix_tuning	48.57 (±**8.23**)	**92.31** (±3.65)	91.61 (±6.97)	75.95 (±5.63)	93.58 (±1.92)	90.27 (±13.72)	80.45 (±8.73)	16.81 (±4.94)	73.69 (±6.73)
lora	55.94 (±21.93)	91.61 (±4.17)	92.53 (±4.31)	76.05 (±8.67)	94.03 (±1.31)	89.57 (±33.03)	78.81 (±10.47)	17.21 (±**1.13**)	74.47 (±10.63)
ia3	50.65 (±13.64)	89.68 (±**2.79**)	90.80 (±7.29)	72.92 (±5.83)	94.22 (±6.04)	89.05 (±21.27)	72.46 (±8.46)	13.51 (±4.08)	71.66 (±8.68)
Full Fine-tuning	**62.80**	90.4	**94.9**	**87.0**	**96.6**	**91.2**	**89.2**	17.73	**86.84**

Table 4: Best performance (± std. dev. across all hyper-parameters) of various supported single adapter methods (refer §3.3.1) applied to `facebook/bart-base`, benchmarked against full fine-tuning. The best results and lowest std. dev. per task are highlighted in **bold**.

PEFT Method	Attribute Name	Range	Added Params. Min	 Max
double_seq_bn	reduction_factor	$\{2, 16, 64\}$	461,088	**14,183,424**
seq_bn	reduction_factor	$\{2, 16, 64\}$	230,544	7,091,712
par_bn	reduction_factor	$\{2, 16, 64\}$	230,544	7,091,712
compacter	reduction_factor phm_dim	$\{4, 16\}$ $\{4, 8\}$	58,816	69,184
prefix_tuning	bottleneck_size prefix_length	$\{32, 128, 512\}$ $\{5, 50, 200\}$	636,704	10,002,944
lora	r	$\{4, 8, 16, 64, 200\}$	147,456	7,372,800
ia3	–	–	**55,296**	55,296

Table 5: Adapter-specific attributes (name as in framework) and their values that we used for grid-search along with the minimum and maximum possible parameters added over the resultant grid. Extreme values are highlighted in **bold**.

B Adapter Evaluation Results

Best task performance for all task–adapter method combinations are presented in Table 3 with RoBERTa as base and model and in Table 4 with BART as base model. All adapter methods provided by *Adapters* are competitive to full fine-tuning. Table 5 gives an overview of the evaluated adapter configurations. Fig. 2 plots best performing learning rate–capacity combinations for evaluated single adapter methods. Lower capacity adapters perform better with higher learning rates.

Fig. 3 plots mean task performance by learning rate for evaluated single adapter methods. For bottleneck adapters, the best-performing learning rate is 10^{-4}. Compacter performs best with a learning rate of 10^{-3}, prefix tuning with $3 \cdot 10^{-4}$, LoRA with $5 \cdot 10^{-4}$, and (IA)³ with $5 \cdot 10^{-3}$. These results align with the learning rates proposed by the respective adapter method authors.

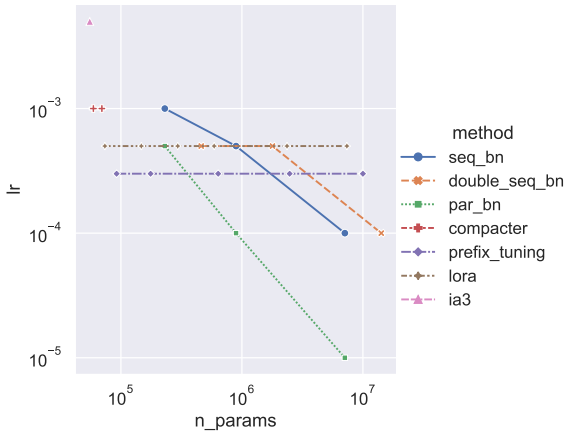


Figure 2: Best performing learning rate–capacity combinations for evaluated single adapter methods. Lower capacity adapters perform better with higher learning rates.

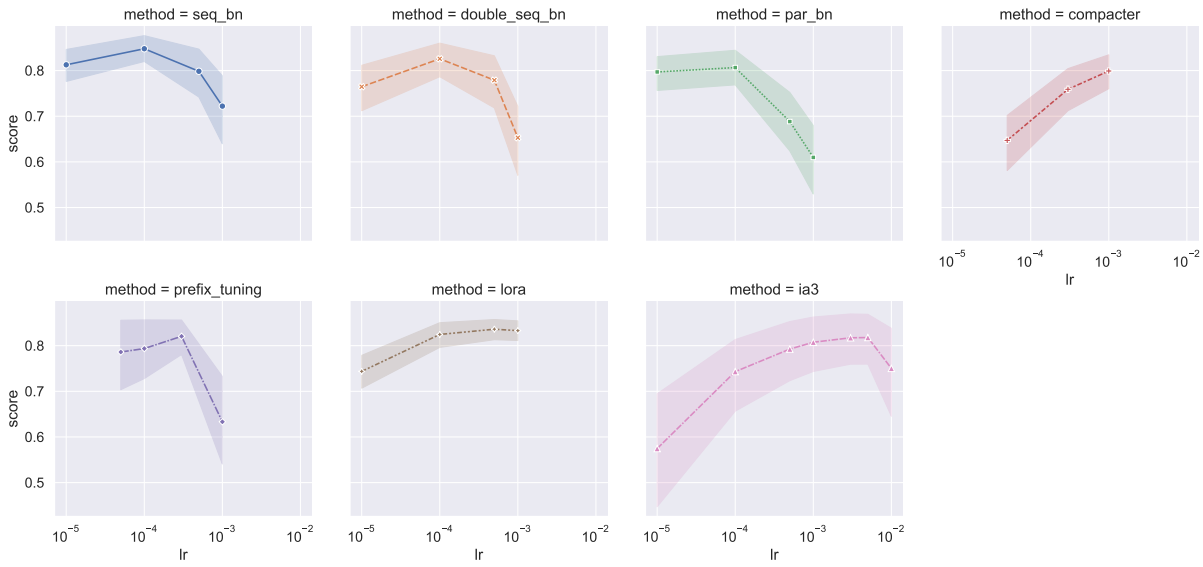


Figure 3: Mean task performance by learning rate for evaluated single adapter methods.