OpenDelta: A Plug-and-play Library for Parameter-efficient Adaptation of Pre-trained Models

Shengding Hu[1,2], **Ning Ding**[1,2], **Weilin Zhao**[1,2], **Xingtai Lv**[1], **Zhen Zhang**[1]
Zhiyuan Liu[1,2,3]*, **Maosong Sun**[1,2]
[1]Dept. of Comp. Sci. & Tech., IAI, BNRIST, Tsinghua University, Beijing
[2]International Innovation Center of Tsinghua University, Shanghai, China
[3]Jiangsu Collaborative Innovation Center for Language Ability, Jiangsu Normal University
hsd20@mails.tsinghua.edu.cn

Abstract

The scale of large pre-trained models (PTMs) poses significant challenges in adapting to downstream tasks due to the high optimization overhead and storage costs associated with full-parameter fine-tuning. To address this, many studies explore parameter-efficient tuning methods, also framed as "delta tuning" in Ding et al. (2022), which updates only a small subset of parameters, known as "delta modules", while keeping the backbone model's parameters fixed. However, the practicality and flexibility of delta tuning have been limited due to existing implementations that directly modify the code of the backbone PTMs and hard-code specific delta tuning methods for each PTM. In this paper, we present OpenDelta [1], an open-source library that overcomes these limitations by providing a plug-and-play implementation of various delta tuning methods. Our novel techniques eliminate the need to modify the backbone PTMs' code, making OpenDelta compatible with different, even novel PTMs. OpenDelta is designed to be simple, modular, and extensible, providing a comprehensive platform for researchers and practitioners to adapt large PTMs efficiently.

1 Introduction

With the rapid development of self-supervised learning methods in the realm of deep learning, especially pre-training techniques (Peters et al., 2018; Devlin et al., 2018; Radford et al., 2018), foundational pre-trained models (Bommasani et al., 2021) (PTMs) have become a common cornerstone for numerous downstream tasks. And as a result, research into large-scale PTMs has flourished.

Nevertheless, the ever-expanding scale of PTMs also poses substantial obstacles in practical use. In traditional model adaptation, all the parameters of the PTMs are optimized for each downstream task, which becomes increasingly impractical as the model scales. Firstly, optimizing all the parameters incurs prohibitive computing and memory consumption; secondly, storing a fine-tuned model instance for each task or experiment significantly amplifies the storage cost.

To address these challenges, researchers have developed parameter-efficient methods for model adaptation. Such methods keep the parameters of the main model fixed and update only a small subset of parameters during adaptation. This approach, known as "*delta tuning*", is described and surveyed in Ding et al. (2022). Different delta tuning methods have been proposed, with varying types and positions of "*delta modules*". For example, Adapter module (Houlsby et al., 2019) is composed of two low-dimensional linear projection layers with an activation function, while LoRA (Hu et al., 2021) module introduces a low-rank decomposition for the weight matrix. BitFit (Zaken et al., 2021), on the other hand, specifies the bias vector in PTMs as the delta modules. The delta module can be applied to different positions (Rücklé et al., 2020; He et al., 2022; Hu et al., 2022) to achieve either better performance or efficiency.

Theoretically, incorporating most delta tuning methods would necessitate restructuring the backbone model, a requirement conventionally achieved through direct code manipulation. While this method may seem simple, it carries several disadvantages. Primarily, it lacks flexibility, as delta modules can theoretically be implemented in various positions, making modifications to each position in the backbone model code a cumbersome task. Additionally, this method is not scalable, as accommodating delta tuning for newly introduced PTMs requires fresh code modifications, posing a challenge for researchers and engineers.

In this paper, we present a novel approach to implement delta tuning methods. Our approach

* corresponding author liuzy@tsinghua.edu.cn
[1]GitHub Repo https://github.com/thunlp/OpenDelta, Demo Video https://rb.gy/qjvpav.

Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics
Volume 3: System Demonstrations, pages 274–281

modifies the backbone model's architecture after it is loaded into the memory. We propose four essential techniques, namely named-based addressing, dynamic tensor re-routing, runtime initialization, and a visualization system. Using these key techniques, we build OpenDelta, an open-source toolkit for delta tuning without modifying the backbone model code. OpenDelta has several key features. Firstly, it is simple to use. Migrating from existing full-parameter training to delta tuning requires as few as three lines of code. For beginners or engineers, we also support automatic delta model construction. Secondly, it is modular, with delta modules implemented as independent sub-modules that can be attached to or detached from the backbone models. This feature allows different delta modules to coexist and cooperate in the same backbone model and serves multiple tasks flexibly. Thirdly, OpenDelta is highly extensible, supporting pre-trained models in a wide range of frameworks, including both official implementations from the Huggingface Library (Wolf et al., 2019) and customized PTMs. It can potentially be used with newly emerged PTMs and integrated with other PTMs' frameworks for efficient training, such as the parallel training framework.

2 Related Work

Our work is related to delta tuning, more specifically, the implementation of delta tuning methods.

Delta Tuning. Delta tuning refers to the parameter-efficient method for tuning a large PTM. Different delta tuning methods (Houlsby et al., 2019; Zaken et al., 2021; Li and Liang, 2021; Hu et al., 2021; Mahabadi et al., 2021; Sung et al., 2022) differ in both the architecture of the delta module and the positions that the delta modules are integrated into the backbone model. Various works have attempted to connect these disparate delta tuning approaches under a unified perspective (He et al., 2022; Ding et al., 2022; Hu et al., 2022). In our work, we draw inspiration from this unified viewpoint and aim to devise a framework that can support different delta tuning methods within the same pipeline. Our library includes the most popular delta tuning methods and is amenable to new methods as they emerge.

Implementation of Delta tuning. Previous implementation frameworks for delta tuning relied on the code modification approach. For example, AdapterHub (Pfeiffer et al., 2020) copies a specific version of Huggingface transformers Library (Wolf et al., 2019) and implement several popular delta tuning methods for a set of pre-defined PTMs. LoRA (Hu et al., 2021) implements a limited library of LoRA linear layers. These methods are model-specific and involve hard-coded implementations, which restrict their usability across various PTMs. In contrast, OpenDelta represents a significant advancement as it requires no code changes to the backbone model, making it highly versatile and broadly applicable.

3 Motivation

In this section, we begin by presenting the unified formulation of delta tuning. Then we underscore a set of crucial characteristics of delta tuning, focusing on the implementation aspect, which emphasizes the pressing need for a novel toolkit to aid in the research and advancement of delta tuning approaches.

3.1 Unified Formulation of Delta Tuning

Although delta tuning is principally not limited to a specific type of neural networks, currently almost all the delta tuning methods are applied to PTMs (Devlin et al., 2019; Liu et al., 2019; Raffel et al., 2019; Brown et al., 2020) with the Transformers architecture (Vaswani et al., 2017). A PTM $\mathcal{M}$ parameterized by Θ is composed of multiple sub-modules m, where the hidden representations $\mathbf{h}$ are passed through the sub-module to produce new hidden representation $\mathbf{h}'$, i.e., $\mathbf{h}' = m(\mathbf{h})$.

The adaptation of a PTM $\mathcal{M}$ to downstream tasks is to update the original parameters Θ into Θ'. In full-parameter fine-tuning, all parameters can be updated, i.e., potentially, $|\Delta\Theta| = |\Theta|$. In contrast, delta tuning only updates a small fraction of parameters, i.e., $|\Delta\Theta| \ll |\Theta|$.

Despite the drastic difference in the specific form of the delta tuning methods, He et al. (2022) unify them into special forms of modifications $\Delta\mathbf{h}$ to the hidden representation $\mathbf{h}$. The $\Delta\mathbf{h}$ is generated by passing a hidden state $\mathbf{h}_\delta$ to a *delta module* m_δ. Formally,

$$\mathbf{h} \leftarrow \mathbf{h} + \Delta\mathbf{h} = \mathbf{h} + m_\delta(\mathbf{h}_\delta), \qquad (1)$$

where $\leftarrow$ denotes a replacement of the original $\mathbf{h}$, and $\mathbf{h}_\delta$ can be the same as or different to $\mathbf{h}$.

3.2 Key Features for Delta Tuning

Several key features of delta tuning methods can be observed from Eq.(1).

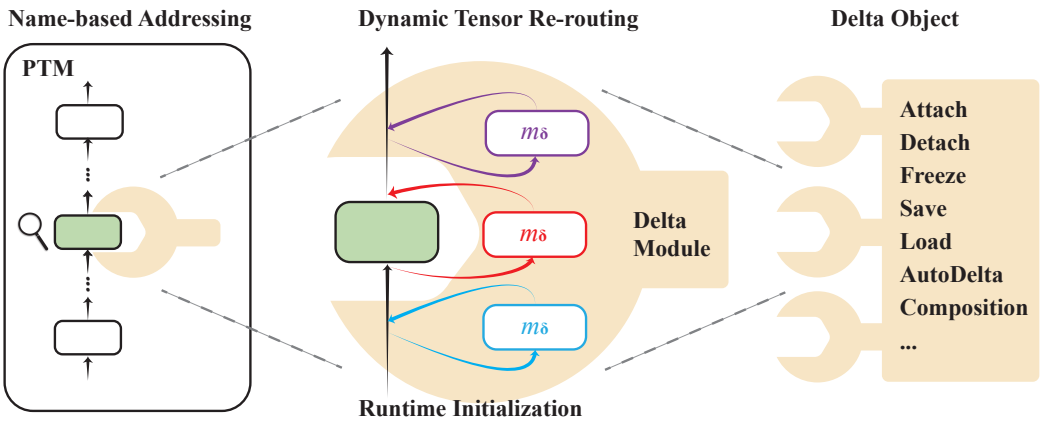

Figure 1: The overall framework of OpenDelta. The construction of delta object happens after the backbone model is loaded.

Tensor Re-routing. The first feature of delta tuning is the ability to redirect the flow of hidden states. In a pre-trained model, the flow of hidden states forms a static graph, with the hidden states serving as nodes and sub-modules acting as transformations on the edges As shown in Eq.(1), the introduction of the edge transformation m_δ redirects node $\mathbf{h}_\delta$ and injects it into another node $\mathbf{h}$, creating a new flow of hidden states that is not present in the original model architecture. The implementation of OpenDelta should achieve such tensor re-routing without hard-coding them.

Flexibility. Eq.(1) allows for the input hidden states and output hidden states to be located at any position in the backbone model $\mathcal{M}$. For example, AdapterDrop (Rücklé et al., 2021) observes that only applying delta modules to the upper half of Transformer layers yields better results than the lower half. The flexibility of applied positions provides remarkable opportunities to explore the potential structure of delta modules (Hu et al., 2022). However, it also presents a challenge for the implementation to be able to achieve flexibility in practice that matches the theoretical framework.

Compositionality. Different delta tuning methods can co-exist or even be combined in the same backbone model (Hu et al., 2022), potentially boosting performance or supporting multitask learning (Pfeiffer et al., 2021). Thus, it is crucial to enable easy and independent implementation of each delta tuning method, while also allowing for the flexible composition of multiple modules.

Dynamism. It is common for the backbone PTM to serve as a central model for multiple tasks in delta tuning. To serve a specific task, delta modules are attached to the backbone model, creating a task-specific expert. When the delta modules are detached, the backbone models revert back to their original function as general language models. This dynamic nature of delta tuning-based task adaptation should be incorporated into OpenDelta.

4 OpenDelta

In light of the aforementioned key features of delta tuning, we present OpenDelta. We will begin by presenting an overview of OpenDelta. Following that, we will delve into the key implementations of this framework.

4.1 Framework

To perform delta tuning, two prerequisites are required: a pre-trained language model $\mathcal{M}$ and the "*modified modules*", which are a user-specified list of sub-modules m_i to which the delta modules should be applied. Our target is to construct a *delta object*. Our objective is to create a delta object, which is a collection of delta modules typically located at various positions within $\mathcal{M}$ and serves as a whole to adapt the PTM to downstream tasks. We follow three steps to create a delta object. Firstly, we use *name-based addressing* to obtain the pointers to the modified modules. Secondly, we construct a delta object comprising uninitialized delta modules. Thirdly, we modify the route of tensors in the modified modules into the delta modules using a *dynamic tensor re-routing* technique. After the updated route of the hidden state is established, we perform *runtime initialization* to initialize the delta object.

After the delta object is constructed, we attach it to the backbone model. Then, we provide a simple functional interface to turn off the gradient

Method	Formulation	Default Positions	Route	Runtime Initialization
LoRA	$m_\delta(\mathbf{h}_{\text{in}}) = \mathbf{h}_{\text{in}}\mathbf{AB}$	Query, Value	Eq.(4)	N
Adapter	$m_\delta(\mathbf{h}_{\text{out}}) = \sigma(\mathbf{h}_{\text{out}}\mathbf{W}_1)\mathbf{W}_2$	ATTN, FFN	Eq.(3)	Y
Bitfit	$m_\delta(\mathbf{h}_{\text{out}}) = \mathbf{b}$	ATTN, FFN, LayerNorm	Eq.(3)	N
Prefix Tuning	$m_\delta(\mathbf{h}_{\text{out}}) = [\text{MLP}(\mathbf{p}); \mathbf{h}_{\text{out}}]$	Key, Value	Eq.(3)	Y

Table 1: Delta tuning methods and their characteristics. Default positions refer to the positions that the delta modules are attached to when no specific sub-modules are designated. $\mathbf{A}$,$\mathbf{B}$,$\mathbf{W}_1$,$\mathbf{W}_2$ are weight matrices, $\mathbf{b}$ is the bias vector. MLP$(\cdot)$ is a multi-layer perception network. $[\cdot;\cdot]$ denotes the concatenation of tensors. σ is the activation function. Runtime Initialization shows whether the implementation uses this technique in OpenDelta.

computation in the backbone models and only compute the gradient of parameters in the delta object. After the training is complete, we provide a simple interface for saving only the delta objects, which significantly reduces the storage requirements for the backbone model.

The overall framework of OpenDelta is shown in Figure 1. Next, we introduce the key implementations that support the construction of delta objects.

4.2 Key Implementations

The above framework is achieved by four key implementations, i.e., name-based addressing, dynamic tensor re-routing, runtime initialization, and visualization system.

Name-based Addressing. Firstly, we need to obtain a pointer to the desired sub-modules which are applied with the delta modules. In practice, we can effectively retrieve the pointer by using the name of the sub-module. Since the sub-modules are organized in a tree structure, we perform a depth-first search to find the sub-modules that match the provided name. This search results in a full path consisting of all the names from the root to the matched sub-module, accurately matching the sub-module. However, directly writing the full path to the sub-modules can be impractical, so we design several simplifications to make addressing easier and more human-readable [2]. One such simplification involves taking advantage of the repetitiveness of transformer layers, which many delta tuning methods address by adding delta modules to the same type of sub-modules in each layer. For example, when users specify `attention`, they likely intend to apply delta modules to the attention sub-modules in all transformer layers. To address this need, we provide a tail-matching mechanism that automatically matches the sub-modules based on their names. For more complex configurations of positions, we allow matching based on regular expressions and web-based selection using our custom-designed web interface.

Dynamic Tensor Re-routing. A fundamental distinction that sets OpenDelta apart from other implementations is its ability to add delta modules without requiring any modifications to the code of the backbone modules. This feature necessitates a dynamic rerouting of tensors through the delta modules and back into the backbone model. To achieve this rerouting, we wrap the original forward function of a sub-module with a wrapper function and replace the original forward function with the wrapper function. To ensure seamless replacement, we utilize a decorator to inherit the original function's attributes, including the I/O, doc string, etc. Within the wrapped function, we implement three distinct routes of the hidden states, taking into account the order of the original sub-module and the delta module. The first route utilizes the input hidden state $\mathbf{h}_{\text{in}}$ of m_i as both the modification target and the input to the delta module. We pass it through the delta module to get the output $m_\delta(\mathbf{h}_{\text{in}})$, and merge it to $\mathbf{h}_{\text{in}}$. Formally,

$$\mathbf{h}_{\text{in}} \leftarrow \mathbf{h}_{\text{in}} + m_\delta(\mathbf{h}_{\text{in}}). \tag{2}$$

The second route employs the output hidden state $\mathbf{h}_{\text{out}}$ of m_i as the modification target:

$$\mathbf{h}_{\text{out}} \leftarrow \mathbf{h}_{\text{out}} + m_\delta(\mathbf{h}_{\text{out}}). \tag{3}$$

The third route leverages the input hidden state $\mathbf{h}_{\text{in}}$ as the input to the delta module, and sets the output hidden state $\mathbf{h}_{\text{out}}$ as the modification target:

$$\mathbf{h}_{\text{out}} \leftarrow \mathbf{h}_{\text{out}} + m_\delta(\mathbf{h}_{\text{in}}). \tag{4}$$

While these three routes do not necessarily encompass all possible relationships between the delta module and the backbone model, they are sufficient to support most popular delta tuning methods (as illustrated in Table 1). However, we

[2] https://opendelta.readthedocs.io/en/latest/notes/namebasedaddr.html

```
model = AutoModel.from_pretrained("bert-base-cased")

+ from bigmodelvis import Visualization
+ Visualization(model).structure_graph()
+ from opendelta import LoraModel
+ delta_model = LoraModel(backbone_model=model, modified_modules=["output.dense"
    , "query"])
+ delta_model.freeze_module(exclude=["deltas", "pooler"], set_state_dict=True)
+ Visualization(model).structure_graph()

trainer.train()
```

Figure 2: An example of basic usage of OpenDelta. '+' sign indicates the additional code needed to enable delta tuning. Note that the visualization can be optional if you are familiar with the backbone model.

remain open to the possibility of incorporating additional routes as needed.

Runtime Initialization. To ensure that weight matrices in the delta module match the hidden states in terms of shape and dimension, we must account for hidden states whose shapes are not specified in the model configuration. In traditional implementations, this requires manually examining the code of the backbone model. However, OpenDelta automates this process by passing a pseudo input through the backbone model, allowing the shapes of the hidden states to be automatically determined as they propagate from the input to the output.

Visualization System. As delta tuning provides flexibility and dynamism, it is essential to ensure the correct construction of delta objects by verifying that delta modules are added as specified. However, direct printing of large pre-trained models results in massive outputs. To address this, we provide a visualization system that leverages repetition in transformer architecture. Specifically, we collapse the repetitive layers and neatly print the parameters' information. With the addition of delta modules to the backbone model, users can easily observe the changes made in the model through visualization. An example of visualization can be seen in Figure 3. As the visualization system is useful beyond delta tuning, it has been separated into an independent package named "`bigmodelvis`" [3].

a

5 Usage

In this section, we provide the use cases of OpenDelta which demonstrate the three characteristics of OpenDelta, i.e., simplicity, modularity, and extensibility.

[3] https://pypi.org/project/bigmodelvis/

```
root
├── embeddings (BertEmbeddings)
│   ├── word_embeddings (Embedding) weight:[28996, 768]
│   ├── position_embeddings (Embedding) weight:[512, 768]
│   ├── token_type_embeddings (Embedding) weight:[2, 768]
│   └── LayerNorm (LayerNorm) weight:[768] bias:[768]
├── encoder (BertEncoder)
│   └── layer (ModuleList)
│       └── 0-11(BertLayer)
│           ├── attention (BertAttention)
│           │   ├── self (BertSelfAttention)
│           │   │   ├── query (Linear) weight:[768, 768] bias:[768]
│           │   │   │   └── lora (LowRankLinear) lora_A:[8, 768] lora_B:[768, 8]
│           │   │   └── key,value(Linear) weight:[768, 768] bias:[768]
│           │   └── output (BertSelfOutput)
│           │       ├── dense (Linear) weight:[768, 768] bias:[768]
│           │       │   └── lora (LowRankLinear) lora_A:[8, 768] lora_B:[768, 8]
│           │       └── LayerNorm (LayerNorm) weight:[768] bias:[768]
│           ├── intermediate (BertIntermediate)
│           │   └── dense (Linear) weight:[3072, 768] bias:[3072]
│           └── output (BertOutput)
│               ├── dense (Linear) weight:[768, 3072] bias:[768]
│               │   └── lora (LowRankLinear) lora_A:[8, 3072] lora_B:[768, 8]
│               └── LayerNorm (LayerNorm) weight:[768] bias:[768]
└── pooler (BertPooler)
    └── dense (Linear) weight:[768, 768] bias:[768]
```

Figure 3: The visualization of the backbone model's status after the LoRA modules are attached.

5.1 Simplicity

Migrating from Fine-tuning. To facilitate the migration from existing full-parameter fine-tuning to delta tuning, only a few lines of code modifications are required, as exemplified in Figure 2. Initially, in the traditional full-parameter fine-tuning, the PTM is loaded from external libraries, such as Huggingface Transformers (Line 1), and train the model (Line 10). To introduce delta tuning, line 3-8 are added and executed. To begin with, an optional step is to visualize the backbone model to identify the target "`modified_modules`". Then, a delta object, such as LoRA, is created and attached to the backbone model. Subsequently, the model parameters, excluding the delta modules and the randomly initialized classification head, are frozen. The "`set_state_dict=True`" parameter is employed to remove the non-trainable parameters from the model checkpoint. Lastly, the sub-modules of the backbone are visualized to verify the successful creation and attachment of the delta modules. An example of the visualization results is depicted in Figure 3.

AutoDelta Mechanism. The implementation of OpenDelta supports highly intricate designs of

```
def multi_task(delta_model, input_text):
    global model # We use the same backbone model across tasks.
    delta_model.attach()
    print(tokenizer.decode(model.generate(input_ids=tokenize(input_text))))
    delta_model.detach()
multi_task("What the commmon career of Newton ad einstein?", spelling_delta)
# >>> "What was the common career of Newton and Einstein?"
multi_task("What was the common career of Newton and Einstein?", topic_delta)
# >>> "The question's topic is science."
multi_task("What was the common career of Newton and Einstein?", question_delta
    )
# >>> "Physicists."
```

Figure 4: Multitask learning via OpenDelta. Due to space limitations, we retain only the core code. For detailed code, please refer to the OpenDelta documentation. Strings after ">>>" demonstrate the output of the model.

delta modules, catering to diverse experimental requirements. Nonetheless, it is desirable to provide a default configuration of delta modules for practitioners who may not be well-versed in the mechanism of delta tuning. However, the naming conventions of sub-modules differ significantly among various backbone models, despite their shared transformer architecture. To tackle this issue, we establish a common name convention and employ a mapping technique to map the model-specific name convention to the common one [4]. This enables the AutoDelta mechanism to be supported seamlessly. Figure 5 exemplifies that, once the type of the delta tuning method is specified, the delta modules will be attached to the backbone model in default positions and with appropriate hyper-parameters. We have listed the default configurations of each delta tuning method in Table 1. Furthermore, the AutoDelta mechanism facilitates the loading of fine-tuned checkpoints of delta modules, without explicit knowledge of the type and hyper-parameters of the delta modules.

```
from opendelta import AutoDeltaModel,
    AutoDeltaConfig
# construct a new delta using the
    default configuration.
delta_config = AutoDeltaConfig.
    from_dict({"delta_type":"lora"})
delta_model = AutoDeltaModel.
    from_config(delta_config,
    backbone_model)
# load the delta checkpoint.
delta = AutoDeltaModel.from_finetuned(
    "save_dir", backbone_model)
```

Figure 5: An example of using AutoDelta mechanism.

5.2 Modularity

The second notable attribute of OpenDelta is modularity. It affords the capacity to independently attach and detach each delta object from the backbone model, thereby providing the possibility of multi-task serving with a single backbone model. Specifically, suppose data pertaining to various tasks are presented sequentially, wherein each data triggers the attachment of a corresponding delta object to the backbone model for processing, and once completed, the delta object is detached. A case that illustrates this functionality is illustrated in Figure 4, where three tasks are process sequentially using a single backbone model.

5.3 Extensibility

Delta tuning is one of the important techniques that enables the use of large PTMs, and as such, we make efforts to ensure its compatibility with other techniques such as model acceleration and multi-GPU training. Specifically, we currently provide support for the BMTrain framework [5] with ZeRO-3 optimization enabled (Rajbhandari et al., 2020). It is also worth noting that we plan to expand our support for additional model-acceleration frameworks in the future.

6 Conclusion

In summary, OpenDelta is a plug-and-play library for delta tuning, offering an intuitive and modular solution to adapt large PTMs using delta tuning without the need for code modifications. The library's user-friendliness, flexibility, and extensibility make it accessible and useful for both researchers and engineers. In the future, we plan to continuously update the library with new delta tuning methods and ensure its compatibility with the latest versions of other major PTMs libraries.

[4] https://opendelta.readthedocs.io/en/latest/notes/unifyname.html

[5] https://github.com/OpenBMB/BMTrain

7 Acknowledgements

This work is supported by the National Key RD Program of China (No.2022ZD0116312), National Natural Science Foundation of China (No. 62236004), Major Project of the National SocialScience Foundation of China (No. 22ZD298).

Limitations

Although we believe that OpenDelta is simple, easy to use, flexible, and extensible since it does not require code modification, it is still limited by many implementation details. For example, some delta tuning methods, such as Prefix Tuning, are limited by theory and can only be used in Attention layers, making them unable to be arbitrarily specified. This is also why we did not use it as an example in this paper. On the other hand, some base models differ significantly from mainstream implementations, making it difficult to use the AutoDelta mechanism. Therefore, we maintain a list of tested models that can use AutoDelta, while other models may still use OpenDelta in a customized manner. Thirdly, while theoretically compatible with acceleration frameworks other than BMTrain, such as Deepspeed, there are some implementation details that currently limit the compatibility of some functions. We will do our best to communicate with the maintainer of those packages to increase compatibility.

Ethical Consideration

In the writing process of this paper, ChatGPT (OpenAI, 2022) was utilized for revision and refinement. However, the authors can guarantee that each sentence in this paper has been thoroughly reviewed and checked to accurately convey the authors' intended meaning.

References

Rishi Bommasani, Drew A Hudson, Ehsan Adeli, Russ Altman, Simran Arora, Sydney von Arx, Michael S Bernstein, Jeannette Bohg, Antoine Bosselut, Emma Brunskill, et al. 2021. On the opportunities and risks of foundation models. *arXiv preprint arXiv:2108.07258*.

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*.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. In *North American Chapter of the Association for Computational Linguistics*.

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, Yujia Qin, Guang Yang, Fuchao Wei, Zonghan Yang, Yusheng Su, Shengding Hu, Yulin Chen, Chi-Min Chan, Weize Chen, et al. 2022. Delta tuning: A comprehensive study of parameter efficient methods for pre-trained language models. *arXiv preprint arXiv:2203.06904*.

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

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*, volume 97 of *Proceedings of Machine Learning Research*, pages 2790–2799. PMLR.

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

Shengding Hu, Zhen Zhang, Ning Ding, Yadao Wang, Yasheng Wang, Zhiyuan Liu, and Maosong Sun. 2022. Sparse structure search for delta tuning. In *In proceedings of NeurIPS*.

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.

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

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

OpenAI. 2022. Chatgpt: Optimizing language models for dialogue.

Matthew E. Peters, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, and Luke Zettlemoyer. 2018. Deep contextualized word representations. In *North American Chapter of the 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. 2020. 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. Association for Computational Linguistics.

Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. 2018. Improving language understanding by generative pre-training.

Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. 2019. Exploring the limits of transfer learning with a unified text-to-text transformer. *ArXiv preprint*, abs/1910.10683.

Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. 2020. Zero: Memory optimizations toward training trillion parameter models. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–16. IEEE.

Andreas Rücklé, Gregor Geigle, Max Glockner, Tilman Beck, Jonas Pfeiffer, Nils Reimers, and Iryna Gurevych. 2020. Adapterdrop: On the efficiency of adapters in transformers. *arXiv preprint arXiv:2010.11918*.

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 EMNLP*, pages 7930–7946.

Yi-Lin Sung, Jaemin Cho, and Mohit Bansal. 2022. Lst: Ladder side-tuning for parameter and memory efficient transfer learning. *arXiv preprint arXiv:2206.06522*.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. 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 5998–6008.

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*.

Elad Ben Zaken, Shauli Ravfogel, and Yoav Goldberg. 2021. Bitfit: Simple parameter-efficient fine-tuning for transformer-based masked language-models. *ArXiv preprint*, abs/2106.10199.