BMInf: An Efficient Toolkit for Big Model Inference and Tuning

Xu Han[1,2*], **Guoyang Zeng**[2,5*], **Weilin Zhao**[1,2*], **Zhiyuan Liu**[1,2,3,4,5†]
Zhengyan Zhang[1,2], **Jie Zhou**[2,5], **Jun Zhang**[1,2], **Chao Jia**[2,5], **Maosong Sun**[1,2,3,4,5†]
[1] Dept. of Comp. Sci. & Tech., Institute for AI, Tsinghua University
Beijing National Research Center for Information Science and Technology
[2] OpenBMB Group [3] Institute Guo Qiang, Tsinghua University
[4] International Innovation Center of Tsinghua University
[5] Beijing Academy of Artificial Intelligence, BAAI
`zenggy@mail.tsinghua.edu.cn` `{hanxu17,zwl19}@mails.tsinghua.edu.cn`
`{liuzy,sms}@tsinghua.edu.cn`

Abstract

In recent years, large-scale pre-trained language models (PLMs) containing billions of parameters have achieved promising results on various NLP tasks. Although we can pre-train these big models by stacking computing clusters at any cost, it is impractical to use such huge computing resources to apply big models for each downstream task. To address the computation bottleneck encountered in deploying big models in real-world scenarios, we introduce an open-source toolkit for **B**ig **M**odel **Inf**erence and tuning (**BMInf**), which can support big model inference and tuning at extremely low computation cost. More specifically, at the algorithm level, we introduce model quantization and parameter-efficient tuning for efficient model inference and tuning. At the implementation level, we apply model offloading, model checkpointing, and CPU-GPU scheduling optimization to further reduce the computation and memory cost of big models. Based on above efforts, we can efficiently perform big model inference and tuning with a single GPU (even a consumer-level GPU like GTX 1060) instead of computing clusters, which is difficult for existing distributed learning toolkits for PLMs. BMInf is publicly released at `https://github.com/OpenBMB/BMInf`.

1 Introduction

Recent years have witnessed the great success of pre-trained language models (PLMs) (Han et al., 2021) in the NLP community. Various techniques of PLMs enable us to train big models containing billions of parameters from large-scale unlabeled corpora in a self-supervised fashion. Up to now, these big models (with billions of parameters like GPT-3 (Brown et al., 2020)) have achieved promising results on various NLP tasks and gained extensive attention from researchers. Despite the success of big models, the massive parameters of these big models also bring challenges to their inference and tuning. Since the pre-training process of big models usually requires to be completed once, the cost caused by massive parameters can be handled by stacking computing resources. However, the inference and tuning process of PLMs depends on specific application scenarios and will frequently use big models for computation. If we still stack devices to speed up the inference and tuning of big models, the cost of time, memory, and even money would become unbearable. In this paper, we introduce a toolkit BMInf, aiming at efficiently performing big model inference and tuning.

As shown in Figure 1, BMInf is built based on a four-level framework, the most important part of which lies in its algorithm level and implementation level. At the algorithm level, we introduce model quantization to compress big models from high-

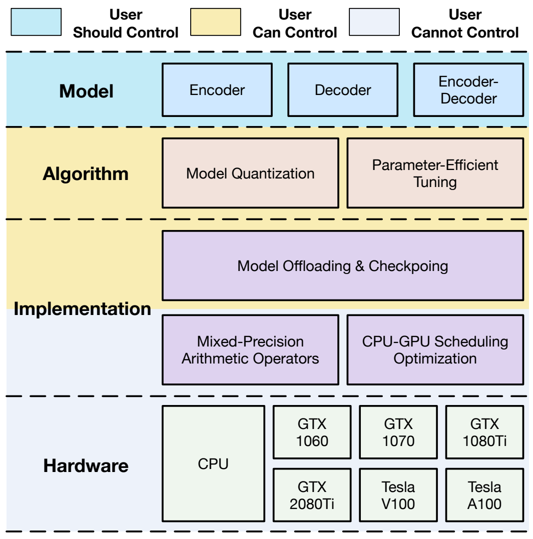

Figure 1: The overall framework of BMInf. To make BMInf convenient for users, the underlying implementation and the hardware adaptation will not be exposed to users, and these modules can be automatically executed.

[*] indicates equal contribution.
[†] indicates corresponding authors.

bit floating-point parameters to low-bit fixed-point ones, which can significantly reduce the memory cost of big models. The faster computation speed of low-bit numbers can also accelerate the computation of big models. Besides model quantization, we also introduce parameter-efficient tuning methods (Ding et al., 2022), which freeze the parameters of big models to reduce the computation and memory cost. By inserting additional learnable modules into big models, parameter-efficient tuning can tune these additional modules to help big models handle specific tasks. Some recent works (Lester et al., 2021; Gu et al., 2021; Hu et al., 2021) have shown that applying parameter-efficient tuning on big models can achieve results comparable to fine-tuning all model weights.

At the implementation level, we implement model offloading and model checkpointing, which can make full use of CPU memory to store massive parameters of big models. Moreover, model offloading and checkpointing can drop parameters and computation graphs during both the forward and backward propagation, which can further save GPU memory to operate more data. For the underlying arithmetic operators, we reimplement the mixed-precision CUDA arithmetic operators, which can better utilize the tensor cores of GPUs to further speed up the computation, especially accelerating the mixed-precision computation in model quantization. Considering model offloading and model checkpointing bring extra CPU-GPU communication to load offloaded model weights, we perform CPU-GPU scheduling optimization to synchronously execute weight loading and model computation. This CPU-GPU scheduling optimization can alleviate the time waiting for weight loading. All of model offloading, model checkpointing, and parameter-efficient tuning can benefit from the scheduling optimization.

Due to the algorithm-level and implementation-level efficiencies, BMInf can work on various GPUs at the hardware level, including both powerful GPUs (e.g. Tesla V100 and Tesla A100) and consumer GPUs (e.g. GTX 1060 and GTX 1080Ti). In Section 4, we will show that BMInf can run models with more than 10 billion parameters on a consumer GPU GTX 1060, which is quite difficult for existing PLM-related distributed toolkits such as Megatron (Shoeybi et al., 2019) and DeepSpeed (Rasley et al., 2020). At the model level, BMInf supports various possible architectures of

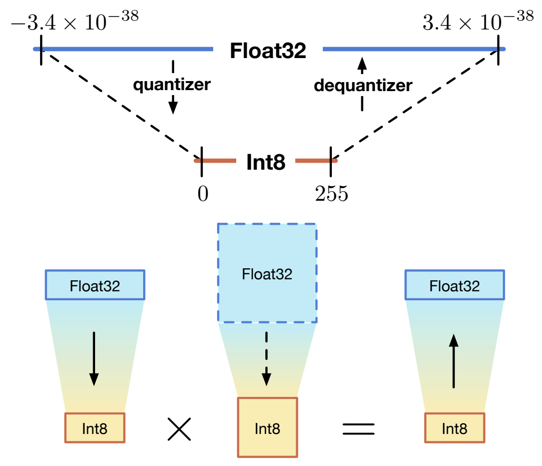

Figure 2: The illustration of model quantization. To balance both efficiency and effectiveness, we use 8-bit fixed-point numbers to represent the weights of all linear layers and higher-bit floating-point numbers (16-bit or 32-bit numbers) to represent hidden states. Here we use 32-bit floating-point numbers as an example. The dotted parts are only used for the low-bit adaptation training.

Transformer-based PLMs, and users can choose their own model architectures for inference and tuning. To make BMInf more convenient for users, the underlying implementation and the hardware adaptation are automatically executed and will not be exposed to users. In the following sections, we will show more details about BMInf, especially at the algorithm level and implementation level.

2 Algorithms to Support the Efficient Inference and Tuning of Big Models

In this section, we briefly introduce how BMInf supports big model inference and tuning in an efficient manner at the algorithm level, including model quantization and parameter-efficient tuning.

2.1 Model Quantization

The massive parameters of big models not only bring a huge amount of computation but also require a lot of memory to store parameters and computation graphs. Therefore, applying model compression is crucial to reduce the computation and memory cost of big models. Since we want big models to maintain generality after model compression, we choose model quantization rather than model pruning and model distillation for our toolkit. The latter two compression approaches are usually used to compress big models for specific tasks and will significantly change the model structure.

Model quantization aims to compress model weights from high-bit floating-point numbers to low-bit fixed-point ones. Typically, PLMs are usually represented with 32-bit or 16-bit floating-point numbers, while their quantized models can be represented with 8-bit, 4-bit, or even 1-bit fixed-point numbers, saving much memory usage. In addition, GPUs have tensor cores specially designed for low-bit numbers, and thus model quantization can also speed up the computation. For Transformer-based PLMs, Zafrir et al. (2019) show that the 8-bit quantization has little impact on the model performance. To further alleviate the performance degradation, Shen et al. (2020) apply mixed-bit quantization where only those parameters with low Hessian spectrum are required to be quantized. Zhang et al. (2020) further utilize knowledge distillation to force low-bit models to imitate high-bit models.

Considering that training a 1-bit or 2-bit Transformer is still challenging due to the significant decrease in model capacity, our toolkit primarily quantizes high-bit (16-bit or 32-bit) models to 8-bit fixed-point ones. The performance of low-bit quantization is highly hardware-related, and those complex quantization mechanisms may only serve specific devices. Therefore, as shown in Figure 2, we apply a simple and effective mixed-bit quantization method, where hidden states are represented with high-bit numbers, while the weights of all linear layers are represented with 8-bit numbers. During the computation, we first quantize the input into 8-bit hidden states and perform computation operations, and then dequantize the output into high-bit states. To make the quantization less impactful on models, we use a small amount of pre-trained data for additional low-bit adaptation training. Specifically, in the low-bit adaptation stage, we still use high-bit numbers to represent model weights, but quantize these weights into low-bit forms for computation (the dotted parts in Figure 2). After the low-bit-adaptation stage, high-bit weights are discarded, and their corresponding low-bit weights are left for inference and tuning.

2.2 Parameter-Efficient Tuning

Vanilla fine-tuning (Radford and Narasimhan, 2018; Devlin et al., 2019) needs to tune all model weights, a mixed-precision PLM with N parameters (Model weights are 16-bit numbers and optimizer states are 32-bit numbers) under this setting would require: (1) N 16-bit weight states and N 16-bit gradient states; (2) N 32-bit master model

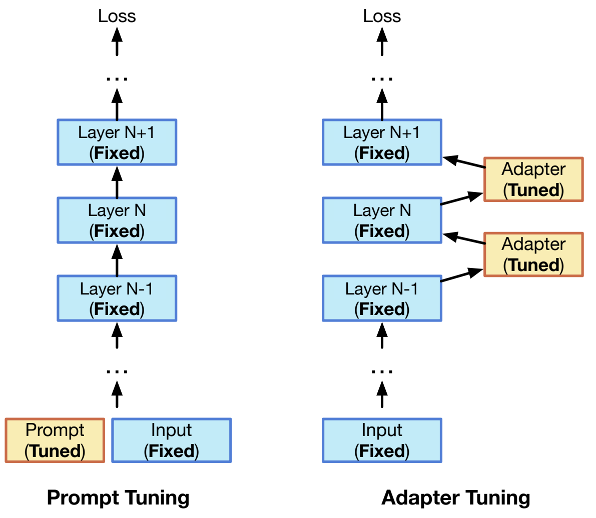

Figure 3: The illustration of parameter-efficient tuning. Here we take prompt tuning and adapter tuning as examples to show how to perform parameter-efficient tuning.

weights; (3) N 32-bit momentum states and N 32-bit variance states for the optimizer. These add up to a total of $16N$ bytes memory consumption. Since a big model has massive parameters, this consumption is too large to compute.

To efficiently tune big models for specific tasks, parameter-efficient tuning (Ding et al., 2022) has been proposed. As shown in Figure 3, the main idea of parameter-efficient tuning is to insert new modules into PLMs and only tune these additional modules, i.e. all PLM weights do not need to be tuned anymore. Under the setting of parameter-efficient tuning, we only need to store the forward and backward information of those tuned modules, which is significantly smaller than tuning all PLM weights. Prompt tuning (Lester et al., 2021; Gu et al., 2021) and adapter tuning (Stickland and Murray, 2019; Houlsby et al., 2019) are two typical parameter-efficient tuning approaches. Prompt tuning aims to better trigger the potential capabilities inside PLMs by only modifying the input. Since PLMs are mostly pre-trained on cloze-style tasks, prompt tuning first inserts several prompt embeddings into the input to adapt all downstream tasks to cloze-style tasks, which can bridge the pre-training and fine-tuning objectives, and then tunes the prompt embeddings to adapt PLMs to specific tasks. Adapter tuning mainly focuses on inserting extra adapter layers into PLMs to help adapt PLMs to handle downstream tasks.

Although freezing all PLM weights has been shown to perform moderately on downstream tasks (Lester et al., 2021), it is still one way to balance time efficiency, memory consumption, and model effectiveness. In fact, some recent parameter-efficient tuning methods (Hu et al., 2021) have achieved comparable results to fine-tuning all model weights. In our toolkit, we provide a unified interface to freeze all weights of big models and compute gradients for those additional modules, which can support various parameter-efficient tuning methods.

3 Implementations to Reduce the Computation and Memory Cost

In this section, we briefly introduce how BMInf reduces the computation and memory cost at the implementation level, including model offloading and model checkpointing, as well as CPU-GPU scheduling optimization. In fact, we also reimplement efficient mixed-precision arithmetic operators to better utilize GPU tensor cores. Since the reimplementation of CUDA operators is too detailed to be described in words, we will not show it here and recommend our readers refer to the source code.

3.1 Offloading and Checkpointing

Although we can exponentially compress big models through model quantization, it is still difficult for the GPU memory to support the storage of model weights and computation graphs. Therefore, we apply model offloading to utilize the CPU memory, which is often very large and cheap. As shown in Figure 4, the main idea of model offloading is to place model weights on the CPU. When the model is computed layer by layer, we load the offloaded weights from the CPU to the GPU for computation. After the computation is completed, we free the loaded weights and computation graphs to save the GPU memory. Since model weights can be divided into small pieces (such a piece is called a phase) to load, model offloading is quite important for running big models using low computing resources.

Although model offloading can well solve the forward propagation of big models, it cannot work for the back propagation, since much of the information used for the backward propagation needs to be computed and preserved in the forward propagation. The consumption of this memory is usually hidden behind the computation graph and often overlooked. In fact, the memory required for the back propagation is also quite huge. Take the matrix multiplication $\mathbf{y} = \mathbf{W}\mathbf{x}$ as an example, it is used almost everywhere in neural networks. Assuming that the gradient of $\mathbf{y}$ has been obtained and denoted as $\frac{\mathrm{d}\mathcal{L}}{\mathrm{d}\mathbf{y}} = \bar{\mathbf{y}}$, we have $\frac{\mathrm{d}\mathcal{L}}{\mathrm{d}\mathbf{x}} = \mathbf{W}^\top\bar{\mathbf{y}}$ and $\frac{\mathrm{d}\mathcal{L}}{\mathrm{d}\mathbf{W}} = \bar{\mathbf{y}}\mathbf{x}^\top$, where $\mathcal{L}$ is the final loss score. That is to say, the memory used for the matrix multiplication cannot be freed immediately after being used, leading to a conflict with model offloading.

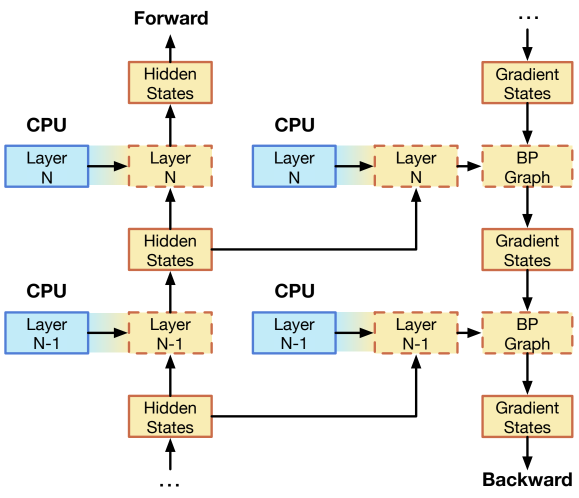

Figure 4: The illustration of model offloading and model checkpointing. All blue parts are stored in the CPU and all yellow ones are in the GPU. The dotted parts are temporary units, whose data and computation graphs will be freed from memory after computation.

To address the issue, we apply model checkpointing, which is also used by existing distributed frameworks such as Megatron and DeepSpeed to accelerate the pre-training of big models. The core of checkpointing is that it allows some of the information used in the back propagation not to be saved in the forward propagation, but to be recomputed in the back propagation. As shown in Figure 4, some hidden states are reserved for the back propagation and all other intermediate results are immediately freed. The reserved information is named “checkpoint”. By dividing big models into several checkpoint-separated phases, freed intermediate results and computation graphs are recomputed as the back propagation passes through these phases, and then released immediately again after obtaining gradient states. Assuming the checkpointing is performed every K operations, this approach reduces the memory footprint to at least $\frac{1}{K}$ of the original one, while only affecting the efficiency by one extra forward propagation time. Generally, offloading phases are consistent with checkpointing

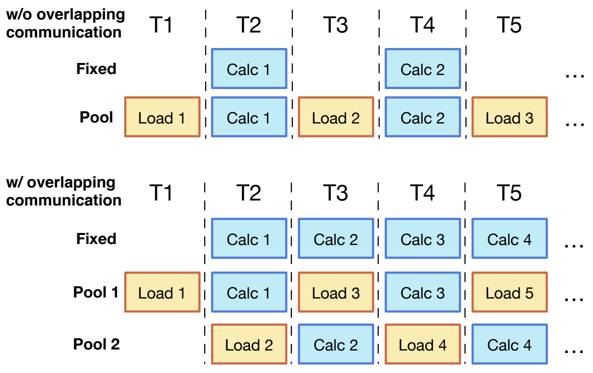

Figure 5: The illustration of overlapping weight loading and model computation. From the figure, we can find that through the CPU-GPU scheduling optimization, the time cost of weight loading can be negligible.

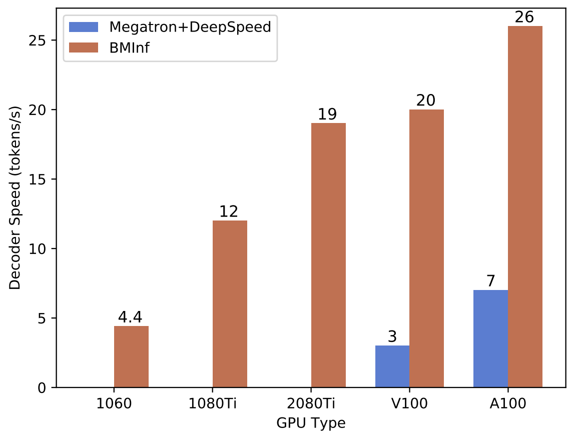

Figure 6: The decoding speed (tokens/s) when performing big model inference with different GPUs.

phases to avoid conflicts.

3.2 CPU-GPU Scheduling Optimization

One negative effect of offloading and checkpointing is that they lead to fragmented memory. In some widely-used deep learning frameworks such as PyTorch and TensorFlow, the GPU memory is allocated dynamically. However, checkpoints are long-lived while those freed and recomputed tensors are short-lived. Suppose the memory allocation is performed in an alternating pattern: long-lived, short-lived, long-lived, short-lived, $\cdots$, this may allow those long-lived tensors to be located in some fragmented regions of the GPU memory, which may affect the efficiency. In the worst case, when a block of M bytes is required, the GPU memory indeed has more than M bytes in total, but a contiguous block of M bytes can never be found for allocation. Meanwhile, model offloading and model checkpointing require frequent communication between CPU and GPU to load model weights, which also brings lots of extra time overhead.

北京环球影城指定单日门票将采用价格滚动制度，即推出淡季日、平季日、旺季日和特定日门票。淡季日门票价格为418元，平季日门票价格为528元，旺季日门票价格为638元，特定日门票价格为688元。
Universal Studios Beijing will adopt a price rolling system, and the prices of low season tickets, mid-season tickets, high season tickets, and special season tickets will be different. The price of low season tickets is 418 RMB, the price of mid-season tickets is 528 RMB, the price of high season tickets is 638 RMB, and the price of special season tickets is 688 RMB.

Table 1: The Chinese text is an inference example of the CPM-2 implemented with BMInf. The underlined tokens are all generated by CPM-2. In this table, we also give the translated English text corresponding to the Chinese text.

To address these issues, we perform a CPU-GPU scheduling optimization. More specifically, we first pre-allocate those long-lived blocks into a contiguous section of the GPU memory ("Fixed" in Figure 5). Then, we pre-allocate two extra memory pools in the GPU to perform weight loading and model computation alternately ("Pool 1" and "Pool 2" in Figure 5). With these two pre-allocated pools, the CPU-GPU communication and the model computation can be synchronously executed, and the CPU-GPU communication time can be completely overlapped in the computation time. Owing to synchronously execution, the time cost of weight loading can be negligible.

4 Evaluation

In this section, we present some evaluation results of big model inference and tuning to show the efficiency of our toolkit BMInf. The following results are based on the model CPM-2 (Zhang et al., 2022). CPM-2 is a Chinese PLM with over 10 billion parameters. Since CPM-2 has an encoder-decoder architecture, it can be used for both text understanding and text generation. The original CPM-2 is implemented with the distributed toolkits DeepSpeed and Megatron, which are currently the most efficient open-source tools for running big models.

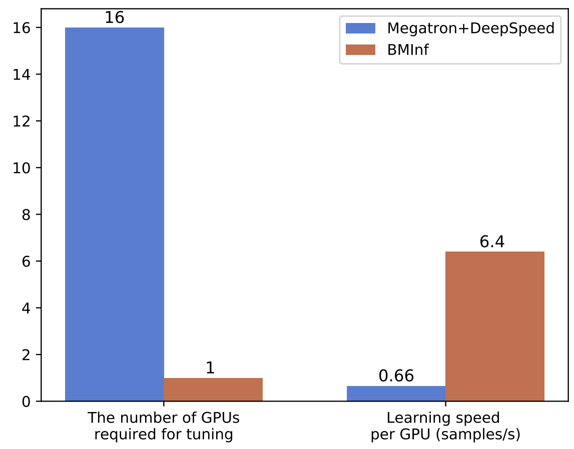

Figure 7: The minimum number of GPUs (Tesla V100) required for prompt tuning (batch size = 32) and the learning speed per GPU (samples/s) when taking minimum GPUs for tuning. Here, the model implemented using Megatron and DeepSpeed has 16-bit floating-point parameters, while the model based on BMInf has 8-bit fixed-point parameters.

4.1 The Results of Big Model Inference

As shown in Figure 6, we can find that models implemented with DeepSpeed and Megatron cannot perform model inference on some consumer GPUs with limited GPU memory, such as GTX 1060 and GTX 1080Ti. For BMInf, even on a GTX 1060 with only 6GB memory units can infer a big model with over 10 billion parameters.

On some powerful GPUs like Tesla V100 and Tesla A100, BMInf achieves $4 \sim 6$ times speedup. In addition to the decoding speed, we also give a case in Table 1, which can intuitively reflect the inference quality of the model implemented with BMInf.

4.2 The Results of Big Model Tuning

In order to evaluate the performance of BMInf on big model tuning, we follow the setting of CPM-2, use CCPM and LCQMC for experiments, and apply prompt tuning to adapt CPM-2 to these two datasets. CCPM is a text classification dataset related to Chinese poems, and LCQMC is a classification dataset of intent similarity.

From Figure 7, we can find that when performing prompt tuning (batch size = 32), the CPM-2 version implemented with DeepSpeed and Megatron requires 16 GPUs, while the version based on BMInf requires only one GPU. For the speed of processing samples, BMInf has achieved nearly 10 times speedup.

Dataset	Model	ACC	GPU
CCPM	FT*	91.6	32
	PT(FP16)*	90.9	16(↓ 50%)
	PT(INT8)	87.4	1(↓ 97%)
LCQMC	FT*	89.2	32
	PT(FP16)*	88.4	16(↓ 50%)
	PT(INT8)	85.3	1(↓ 97%)

Table 2: The comparison between fine-tuning (FT) and prompt tuning (PT). "PT(INT8)" is implemented based on the model quantization of BMInf. "*" means the result is from the CPM-2 paper (Zhang et al., 2022). "ACC" means the accuracy of models (%) and "GPU" means the minimum GPU number required for tuning. "↓" indicates a percentage decrease in the minimum number of GPUs required for tuning.

From Table 2 we can find that model quantization still affects the model performance to a certain extent. The reason is that the models with more parameters are more susceptible to the low-bit variance brought by the quantization methods. Although 8-bit quantization has been demonstrated the little impact on those models with millions of parameters, how to robustly quantify those big models with billions of parameters remains us an future work.

5 Conclusion and Future Work

In this paper, we introduce an efficient toolkit BMInf to provide a way to use large-scale PLMs. By applying model quantization, parameter-efficient tuning, model offloading, model checkpointing, CPU-GPU scheduling optimization, as well as the reimplementation of mixed-precision arithmetic operators, BMInf can perform big model inference and tuning with less than 1/30 of the GPU memory and 10 times speedup, as compared with existing open-source distributed toolkits for pre-training and fine-tuning PLMs.

In the future, our work to improve BMInf will focus on the following three directions:

(1) At the model level, we will gradually support more models;

(2) At the algorithm level, we will continue to improve our model quantization methods to achieve better performance, and work with other toolkits such as OpenPrompt (Ding et al., 2021) to explore more effective ways to tune big models;

(3) At the implementation level, we will provide long-term maintenance for this toolkit.

We hope this toolkit can help researchers utilize big models for their own works and advance the adaption of big models in the NLP community.

Acknowledgements

This work is supported by the National Key R&D Program of China (No. 2020AAA0106502), Institute Guo Qiang of Tsinghua University, Beijing Academy of Artificial Intelligence (BAAI), and International Innovation Center of Tsinghua University.

References

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, et al. 2020. Language models are few-shot learners. In *Proceedings of NeurIPS*.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of NAACL-HLT*, pages 4171–4186.

Ning Ding, Shengding Hu, Weilin Zhao, Yulin Chen, Zhiyuan Liu, Hai-Tao Zheng, and Maosong Sun. 2021. Openprompt: An open-source framework for prompt-learning. *arXiv preprint arXiv:2111.01998*.

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*.

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

Xu Han, Zhengyan Zhang, Ning Ding, Yuxian Gu, Xiao Liu, Yuqi Huo, Jiezhong Qiu, Liang Zhang, Wentao Han, Minlie Huang, et al. 2021. Pre-trained models: Past, present and future. *AI Open*.

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 ICML*, pages 2790–2799.

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 arXiv:2106.09685*.

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

Alec Radford and Karthik Narasimhan. 2018. Improving language understanding by generative pre-training. *OpenAI Blog*.

Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. 2020. DeepSpeed: System optimizations enable training deep learning models with over 100 billion parameters. In *Proceedings of KDD*, pages 3505–3506.

Sheng Shen, Zhen Dong, Jiayu Ye, Linjian Ma, Zhewei Yao, Amir Gholami, Michael W Mahoney, and Kurt Keutzer. 2020. Q-bert: Hessian based ultra low precision quantization of bert. In *Proceedings of AAAI*, pages 8815–8821.

Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. 2019. Megatron-LM: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*.

Asa Cooper Stickland and Iain Murray. 2019. Bert and pals: Projected attention layers for efficient adaptation in multi-task learning. In *Proceedings of ICML*, pages 5986–5995.

Ofir Zafrir, Guy Boudoukh, Peter Izsak, and Moshe Wasserblat. 2019. Q8bert: Quantized 8bit bert. In *Proceedings of NeurIPS*.

Wei Zhang, Lu Hou, Yichun Yin, Lifeng Shang, Xiao Chen, Xin Jiang, and Qun Liu. 2020. Ternarybert: Distillation-aware ultra-low bit bert. In *Proceedings of EMNLP*, pages 509–521.

Zhengyan Zhang, Yuxian Gu, Xu Han, Shengqi Chen, Chaojun Xiao, Zhenbo Sun Yuan Yao, Fanchao Qi, Jian Guan, Pei Ke, Yanzheng Cai, et al. 2022. Cpm-2: Large-scale cost-effective pre-trained language models. *AI Open*.