УДК 004.85

Methods for Optimizing the Training and Fine-Tuning Large Language Models

A. V. Samonov — PhD in Engineering. Research interests: system analysis, computer science, system and software engineering, methods and means of information security. E-mail: a.samonov@mail.ru

Mozhaisky Military Aerospace Academy, 13 Zhdanovskaya str., St. Petersburg, 197198, Russia

For citation: Samonov A. V. Methods for Optimizing the Training and Fine-Tuning Large Language Models // Intellectual Technologies on Transport. 2024. No. 3 (39). Pp. 5–12. DOI: 10.20295/2413-2527-2024-339-5-12

Abstract. *The main problematic issues in the development and specialization of LLM are: catastrophic forgetting, the risk of overfitting, hallucinations, incorrect interpretations, incorrect processing of exceptional situations as well as exceptionally high performance requirements for the computing tools used in this case. The purpose of the study is to select and develop methods for optimizing the training and fine-tuning process LLM, providing a significant reduction in the computing resources required for this. To achieve this goal, it is proposed to use the following methods of optimizing LLMs and their learning algorithms: LoRA and QLoRA, Batch size choice, Gradient Accumulation, Gradient Checkpoint, Mixed precision training, FlashAttention-2. To obtain a cumulative positive effect when using these methods together, it is necessary to perform a number of practical experiments. When setting up LLM learning hyperparameters, you should first determine which package size gives the best results, and then choose adequate methods to optimize the computing resources used. The application of the presented methods will increase the efficiency of using computing resources when training and fine-tuning large language models and will reduce the time and financial costs necessary for this.*

Keywords: *fine-tuning, gradient accumulation, graphics processing unit, Large Language Model, Low-Rank Adaptation, mixed precision*

Introduction

The current stage of global development is characterized by the active introduction of artificial intelligence technologies into industry, science, education and other spheres of economic and social life, vivid examples of which are deep learning methods and generative artificial intelligence. Large language models (LLM) created with their help are capable of processing and creating texts, understanding and synthesizing speech, images, generating program code, solving analytical, mathematical and other non-trivial tasks. The main problematic issues in the development and specialization of LLM are: catastrophic forgetting, the risk of overfitting, hallucinations, incorrect interpretations, correct handling of exceptional situations, ensuring the integrity of models, exceptionally high performance requirements for computing tools used in LLM training.

This article discusses methods for optimizing the training and fine-tuning process LLM, which ensure a significant reduction in the computing resources required for this with minimal loss of quality of the created models.

Stages of development of specialized systems based on large language models

The technological chain of the process of creating specialized software systems based on large language models includes three main stages: the development of a basic model, preliminary specialization and fine-tuning to solve problems in a specific subject area.

At the initial stage, the model is trained on unstructured and unlabeled data. The main data sources that were used in the development of most modern LLMs are: Wikipedia, Common Craw, BooksCorpus (a collection of book texts), OpenWebText (a set of articles from the Internet). The result is a basic general purpose LLM. Examples of such models are GPT 4, GPT 3.5, Claude2, Gemini, Falcon, Llama, T5 [1].

At the second stage, the LLM is finalized through self-study on specially prepared data, which adjusts the model to solve problems of a certain class. At the third stage, such models undergo additional training with reinforcement based on expert feedback (RLHF, Reinforcement Learning from Human Feedback). Such open-source LLMs as Llama, Vicuna 13B, Mixtral 7B, T5 can be used to create specialized LLMs [1, 2]. As a result, domain-specific AI systems are being created, designed to solve certain tasks in specific areas of application.

LLM training and fine-tuning require high-performance computers, large amounts of RAM and disk memory, and graphics processing unit — GPU. During LLM training, GPU memory is used to store model weights, optimizer states, gradients (parameter derivatives), and direct activations stored to calculate gradients. When calculating weight coefficients in fp32 format, 4 bytes of GPU memory are required per LLM parameter. Thus, to load a model with three billion parameters, 12 GB of memory (4 bytes × 3000000) will be required. In addition, an additional 6 GB (2 bytes × × 3000000) will be required to store the states of the AdamW 8-bit optimizer, and 12 GB of memory (4 bytes × × 3000000) will be required to accommodate gradients. Thus, it turns out that to train a model with three billion parameters, at least 30 GB of RAM will be required.

The main characteristics of the GPU are: memory capacity, performance on special floating-point tasks, scalability, virtualization support, power consumption and price. Depending on the characteristics, the cost of the GPU varies from hundreds of thousands (4090 ADA 24 GB 3.5 Slot FP16 = 83 TFLOPS) to several million rubles (H100 Hopper 80 GB, FP16 = 204.9 TFLOPS) [3, 4]. At the same time, according to estimates [2], to train LLM with 3 billion parameters, it is necessary to use 128 GPU A100 40 GB for 7 days. Given that modern LLMs have tens and hundreds of billions of parameters, it becomes obvious that there is an urgent need to develop methods to optimize the processes of their training, configuration and use.

The research task

In connection with the above, it becomes obvious that there is an urgent need to develop and apply less expensive methods and tools for training and configuring LLM that can implement these processes without significantly reducing the quality of the intelligent systems created at the same time. The following subsections of the article provide a brief description of such methods and tools, as well as suggestions for their optimal use.

Method for fine-tuning large language models

In order to reduce the performance requirements of computing tools used in LLM training, methods Parameter-Efficient Fine-Tuning (PEFT) have been developed and are being used. A description of modern methods and tools for fine-tuning LLM is presented in [5–9]. The software implementation of these methods is presented in the *peft* library on site *huggingface.co*.

Methods for fine-tuning effective LLM parameters, unlike full model tuning, provide training for only a small set of parameters (PEFT), which can be a subset of existing model parameters or a set of added parameters. Usage PEFT methods allows you to reduce training time, reduce the cost of computing and storing models, reduce the risks of overtraining, overcome catastrophic forgetting, correctly handle exceptional situations, and ensure ease of deployment and transfer to other devices.

Currently, the most promising methods of fine tuning are: LoRA (Low-Rank Adaptation) and QLoRA (Quantization-Aware LoRA). The LoRa method focuses on changing the weights of only certain layers and parameters of the basic LLM, focusing on those that are most useful for solving problems of this class. This is achieved by applying matrices with significantly lower rank than the matrices of the basic model to adjust the weights.

LoRa applies only to transformer query and value matrices, which means that the multilayer perceptron is frozen and only attention weights are adapted. The loss function is optimized by passing the gradient through the frozen model to the adapters. The formula describing the LoRA method in tensor notation has the following form:

$$W0 + \Delta W = W0 + BA,$$

where $W0$ is the weight matrix of the pre-trained model;

ΔW is the updated and added weight coefficients during the adaptation of the original model;

$A \in Rr \times k$ is a matrix of size $r \times k$, the elements of which are random variables corresponding to the normal distribution law $N(\mu, \sigma^2)$, where $\mu = 0$ (the average value of the value), σ is the standard deviation; $B \in Rd \times r$ is a matrix of size $d \times r$, the elements of which are assigned zeros at the initial stage of training.

An important advantage of LoRa is the ability to use the same model for different tasks by replacing the weights in matrices A and B, reducing the amount of memory needed to store different models.

The QLoRA method (LoRA with quantization) is designed to deploy models in environments with limited resources. It allows you to significantly reduce the requirements for the necessary amounts and performance of GPU and CPU, as well as computing power, for deploying and configuring models. QLoRA is a modification of the LoRA method by quantifying the model parameters, i.e. reducing the accuracy of the weighting coefficients, while maintaining the necessary correctness and performance.

The number of parameters is determined by the rank and shape of the original weights. In practice, trainable parameters vary as low as 0.1 % to 1 % of all the parameters. As the number of parameters needing fine-tuning decreases, the size of gradients and optimizer states attached to them decrease accordingly. Thus, the overall size of the loaded model reduces. For example, the Llama 2 7B model parameters could be loaded in int8 (1 byte), with 1 GB trainable parameters loaded in fp16 (2 bytes). Hence, the size of the gradient (fp16), optimizer states (fp32), and activations (fp32) aggregates to approximately 7–9 GB. This brings the total size of the loaded model to be fine-tuned to 15–17 GB, as illustrated in Fig. 1.

Thus, thanks to the use of LoRA, it was possible to reduce the amount of memory required to configure Llama 2 7B by 4 times.

Methods for optimizing the learning process and configuring large language models

In order to increase the efficiency of using computing resources when training large language models to solve problems in a certain subject area, the following approaches and methods are currently used: Gradient Accumulation, Gradient Checkpoint, Mixed precision training, FlashAttention-2 [10–12].

The Gradient Accumulation method provides the calculation of gradients in smaller increments instead of calculating them for the entire batch at once. Iterative calculation of gradients is performed in small batches by performing forward and reverse

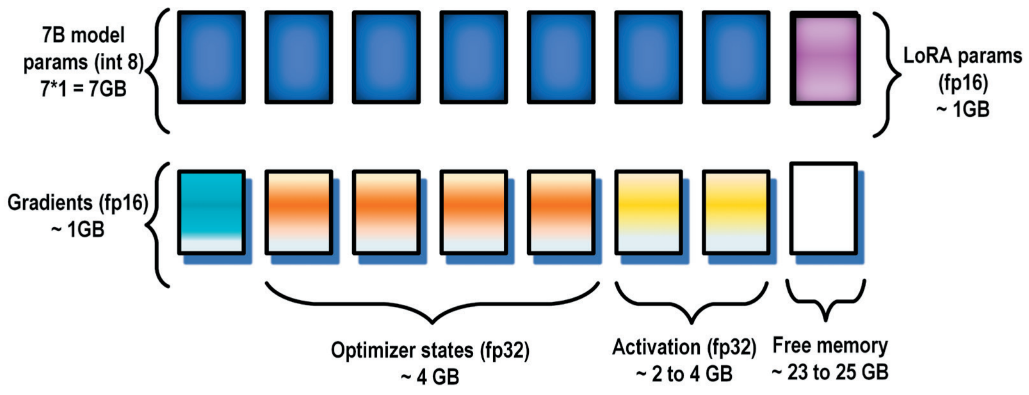
7B model params (int 8) 7*1 = 7GB
LoRA params (fp16) ~ 1GB
Gradients (fp16) ~ 1GB
Optimizer states (fp32) ~ 4 GB
Activation (fp32) ~ 2 to 4 GB
Free memory ~ 23 to 25 GB

Fig. 1. Schematic showing an example of memory footprint of LoRA fine tuning with Llama 2 7B model

passes through the model and accumulating gradients in the process. Once a sufficient number of gradients have been accumulated, the model is optimized. Using this method, the effective packet size can be increased beyond the limits imposed by the amount of memory of the GPU. At the same time, it should be borne in mind that additional passes forward and backward, implemented in the process of gradient accumulation, can slow down the learning process. To use this method when training a model, *the gradient_accumulation_steps* argument must be included in the *TrainingArguments* configuration file:

*training_args = TrainingArguments(per_device_train_batch_size=1, gradient_accumulation_steps=4, **default_args)*

Using the "Gradient Accumulation" method allows you to maximize the use of GPU resources. Examples and results of the application of the "*gradient accumulation*" method are presented in [12].

To save all forward pass activations, significant amounts of memory must be allocated to calculate gradients during the reverse pass. If you do not save the activations, then their re-calculation during the reverse passage through the graph of the model will lead to significant computational costs and slow down the learning process. The "*Gradient Checkpointing*" method offers a compromise between these two approaches and preserves strategically important activations for the entire computational graph at certain control points. Due to this, only a part of the activations needs to be calculated again. To use this method, the "*gradient_checkpointing=True*" argument must be included in the TrainingArguments configuration file. The use of this method increases the efficiency of memory usage, but slows down learning by about 20 % [12].

Using the "*Mixed precision training*" method, the efficiency of the model training process is increased by using lower precision numerical formats for certain variables. Most models use 32-bit floating-point precision (fp32 or float32) to represent and process variables.

However, not all variables require such a high level of accuracy to achieve good results. By reducing the precision of some variables, for example, to 16-bit floating point values (fp16 or float16), calculations can be accelerated. The main advantage of mixed-precision learning is the storage of half-precision activations (fp16). Although gradients are also calculated with half accuracy, they are converted back to full accuracy during the optimization stage. Therefore, there is no memory saving in this case.

Thus, learning with mixed accuracy, on the one hand, leads to faster calculations, and on the other hand, it can lead to an increase in the amount of GPU memory used, especially with small packet sizes. This is due to the fact that the model is now present on the GPU with both 16-bit and 32-bit precision, i.e. 1.5 times more than the original model. The scheme of the algorithm mixed precision method, used for calculating LLM parameters is shown in Fig. 2.

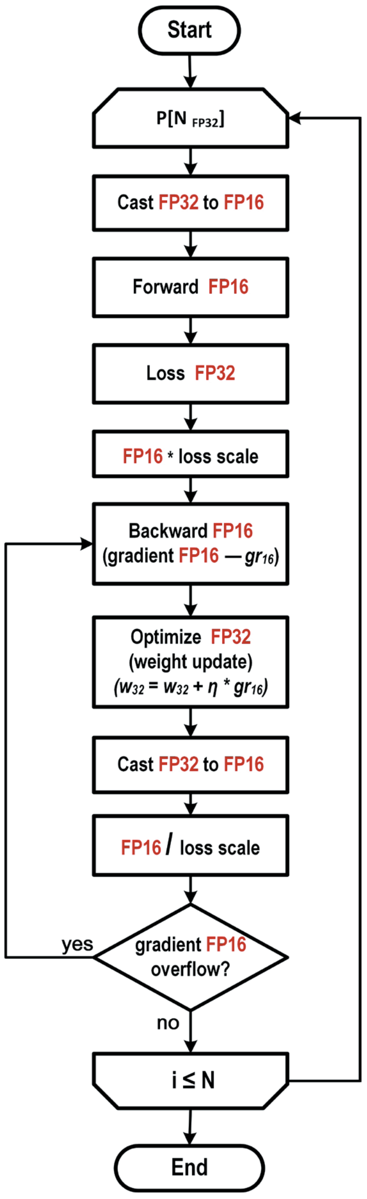

Fig. 2. Scheme of the algorithm mixed precision method

The parameters P[N_{FP32}] received at the input of the algorithm in FP32 format are converted to FP16 format. The loss level is set to FP32. During backward computation, the value is multiplied by the loss scale to avoid overflow due to a small gradient value. A parameter in FP16 format is used to calculate the gradient, and the result is converted to FP32. Then the value is divided by the loss scale to restore the multiplied gradient. The optimizer checks if the gradient is overflowing. If yes, the optimizer skips the update. If not, the optimizer uses FP32 to update the initial parameters. To use the mixed precision method, you must set the "*fp16=True*" parameter in the *Training Arguments* configuration file.

FlashAttention-2 is a faster and more efficient implementation of the standard attention mechanism, which can significantly speed up logical inference due to:

- additional parallelization of attention calculations along the length of the sequence;
- separation of work between GPU threads to reduce data exchange and read/write operations in shared memory between them.

FlashAttention-2 can only be used if the model format is fp16 or bf16. To use the FlashAttention-2 method, the "*attn_implementation="flash_attention_2"*" parameter must be included in the model description. FlashAttention-2 can be combined with other optimization methods such as quantization. Below is an example of using this method in combination with 8-bit quantization:

```
# load in 8bit
model = AutoModelForCausalLM.from_pretrained
(model_id,
load_in_8bit=True,
attn_implementation"flash_attention_2")
```

Reducing the requirements for the amount of GPU memory required for LLM training can be achieved by choosing the optimal optimizer. Below are the GPU memory requirements required by three different optimizers when learning LLM with 3 billion parameters [10]:

- the standard AdamW optimizer will require 24 GB of GPU memory, since 8 bytes are used for each parameter (8*3 => 24 GB);
- adafactor optimizer will require more than 12 GB, because a little more than 4 bytes are used for each parameter;
- an 8-bit quantized BNB optimizer will use only (2*3) 6 GB if all the states of the optimizer are quantized.

Adafactor does not store moving averages for each element in weighting matrices. Instead, it stores aggregated information (sums of moving averages by rows and columns), which significantly reduces the memory used.

However, compared to Adam, Adamfactor may have slower convergence in some cases. Which optimizer will be used is determined in the *TrainingArguments* configuration file using the "*optim="adafactor"*" parameter.

In combination with other approaches (gradient accumulation, gradient checkpoint detection, and training with mixed accuracy), you can get a three-fold reduction in the size of the required memory while maintaining bandwidth. However, as mentioned earlier, the convergence coefficient of Adafactor may be worse than Adam.

An example of program implementing a learning cycle of a large language model

This section provides an example of a program that uses the methods described above to training basic LLM using the functions of the Accelerate library of the Pytorch framework. The configuration file includes a description of the following parameters of the learning process:

```
training_args = TrainingArguments
(per_device_train_batch_size=1,
gradient_accumulation_steps=4,
gradient_checkpointing=True
fp16=True,
**default_args)
```

A fragment of a program that implements a learning cycle using the Accelerator module:

```
from accelerate import Accelerator
from torch.utils.data.dataloader import DataLoader
dataloader = DataLoader(ds, batch_
size=training_args.per_device_train_batch_size)
if training_args.gradient_checkpointing:
model.gradient_checkpointing_enable()
accelerator = Accelerator(fp16=training_args.fp16)
model, optimizer, dataloader = accelerator.
prepare(model, adam_bnb_optim, dataloader)
```

First, we load the training dataset using the DataLoader data loader. To optimize the learning process of the model, we use the gradient_checkpointing_enable() method and the mixed precision learning mode — fp16. In the call to the prepare method, it is determined that the data loader will be distributed between processes if we use several GPUs, and the 8-bit adam_bnb_optim optimizer will be used for training. Below is a fragment of the program that implements the main learning cycle:

```
model.train ()
for step, batch in enumerate (dataloader, start=1):
loss = model(**batch).loss
loss = loss / training_args.gradient_accumulation_steps
accelerator.backward(loss)
if step % training_args. gradient_accumulation_steps == 0:
optimizer.step()
optimizer.zero_grad()
```

Step is the number of accumulation steps, *batch* is the batch size assigned in the dataloader. The error back propagation method is performed using the accelerator *function.backward(loss)*. Gradient accumulation is performed as follows: we normalize the losses, get the average value at the end of accumulation, and as soon as we have enough steps, we start the model learning process – optimizing the model weights using the *optimizer.step()*. Optimizer *method.zero_grad()* – resets the gradients of all optimized tensors (weight coefficients).

The methods discussed in this article allow you to reduce the learning time and increases efficiency the use of GPU and CPU. To obtain a cumulative positive effect when using these methods together, it is necessary to plan and perform a number of practical experiments. When setting up LLM learning hyperparameters, you should first determine which package size gives the best results, and then choose adequate methods to optimize the computing resources used. Examples and results of such studies and experiments are presented in [12–15].

Conclusion

This article presents methods to reduce the requirements for the number and performance of computing tools necessary for teaching large language models by optimizing models and algorithms for their training, as well as methods to increase the efficiency of using available computing resources when they are fine-tuned to solve problems in a specific subject area. The most significant results presented in the article, from the point of view of scientific novelty and practical significance, are:

1. Methodological recommendations for the use of LoRA (Low-Rank Adaptation) and QLoRA (Quantization-Aware LoRA) methods for fine-tuning large language models, which provide a significant (by an order of magnitude or more) reduction in computing performance requirements.

2. Algorithmic and software support of LLM learning processes using the methods Gradient Accumulation, Gradient Checkpoint, Mixed precision training and FlashAttention-2, which provides an increase in the efficiency of using available computing resources when teaching and using large language models to solve problems in a specific subject area. With the complex and correct application of these methods, it is possible to obtain a threefold reduction in the size of the required memory while maintaining bandwidth and the quality of the results obtained.

REFERENCES

1. A Survey of Large Language Models / W. Zhao [et al.] // ArXiv. 2023. Vol. 2303.18223. 124 p. DOI: 10.48550/arXiv.2303.18223

2. Down to Scale Up: A Guide to Parameter-Efficient Fine-Tuning / V. Lialin [et al.] // ArXiv. 2023. Vol. 2303.15647. 21 p. DOI: 10.48550/arXiv.2303.15647

3. Matrix Multiplication Background User's Guide // NVIDIA Documentation Hub. URL: http://docs.nvidia.com/deeplearning/performance/dl-performance-matrix-multiplication (accessed 26 Mar 2024).

4. Bekman S. Benchmarking Transformers with HF Trainer on a Single A100 40GB // Github. URL: http://github.com/huggingface/transformers/issues/15026 (accessed 26 Mar 2024).

5. LORA: Low-Rank Adaptation of Large Language Models / E. Hu [et al.] // ArXiv. 2021. Vol. 2106.09685. 26 p. DOI: 10.48550/arXiv.2106.09685

6. LLaMA-Adapter: Efficient Fine-Tuning of Language Models with Zero-Init Attention / R. Zhang [et al.] // ArXiv. 2023. Vol. 2303.16199. 22 p. DOI: 10.48550/arXiv.2303.16199

7. Delta Tuning: A Comprehensive Study of Parameter Efficient Methods for Pre-Trained Language Models / N. Ding [et al.] // ArXiv. 2022. Vol. 2203.06904. 49 p. DOI: 10.48550/arXiv.2203.06904

8. QA-LoRA: Quantization-Aware Low-Rank Adaptation of Large Language Models / Y. Xu [et al.] // ArXiv. 2023. Vol. 2309.14717. 16 p. DOI: 10.48550/arXiv.2309.14717

9. QDyLoRA: Quantized Dynamic Low-Rank Adaptation for Efficient Large Language Model Tuning / H. Rajabzadeh [et al.] // ArXiv. 2024. Vol. 2402.10462. 6 p. DOI: 10.48550/arXiv.2402.10462

10. Methods and Tools for Efficient Training on a Single GPU // Hugging Face Community. URL: http://huggingface.co/docs/transformers/perf_train_gpu_one (accessed 26 Mar 2024).

11. Goodfellow I., Bengio Y., Courville A. Optimization for Training Deep Model // Deep Learning. Cambridge (MA): MIT Press, 2016. Pp. 267–320.

12. Bekman S. Benchmarking Transformers with HF Trainer on RTX-309 // Github. URL: http://github.com/huggingface/transformers/issues/14608 (accessed 26 Mar 2024).

13. Linear/Fully Connected Layers User's Guide // NVIDIA Documentation Hub. URL: http://docs.nvidia.com/deeplearning/performance/dl-performance-fully-connected (accessed 26 Mar 2024).

14. Exploring Parameter-Efficient Fine-Tuning Techniques for Code Generation with Large Language Models / M. Weyssow [et al.] // ArXiv. 2023. Vol. 2308.10462. 23 p. DOI: 10.48550/arXiv.2308.10462

15. PTraining FP8 Large Language Models / H. Peng [et al.] // ArXiv. 2023. Vol. 2310.18313. 23 p. DOI: 10.48550/arXiv.2310.18313.10.48550/arXiv.2310.18313

Received: 18.07.2024
Accepted: 13.08.2024

Методы оптимизации процесса обучения и тонкой настройки больших языковых моделей

А. В. Самонов — канд. техн. наук. Область научных интересов: системный анализ, информатика, математическое и имитационное моделирование и разработка сложных программно-технических систем, методы и средства обеспечения информационной безопасности. E-mail: a.samonov@mail.ru

Военно-космическая академия имени А. Ф. Можайского, Россия, 197198, Санкт-Петербург, ул. Ждановская, 13

Для цитирования: Самонов А. В. Методы оптимизации процесса обучения и тонкой настройки больших языковых моделей // Интеллектуальные технологии на транспорте. 2024. № 3 (39). С. 5–12. DOI: 10.20295/2413-2527-2024-339-5-12

Аннотация. *Основными проблемными вопросами при разработке и специализации больших языковых моделей (Large Language Model — LLM,) являются катастрофическое забывание, риск переобучения, галлюцинации, некорректная обработка исключительных ситуаций, а также исключительно высокие требования к производительности используемых при этом вычислительных средств. Целями исследования являются выбор и разработка методов оптимизации процесса обучения и настройки LLM, обеспечивающих существенное снижение необходимых для этого вычислительных ресурсов. Для достижения* *данной цели предложено использовать следующие методы оптимизации LLM и алгоритмов их обучения: LoRA и QLoRA, Batch size choice (выбор оптимального размера пакета), Gradient Accutulation (накопление градиента), Gradient Checkpointing (контрольные точки градиента), Mixed precision training (смешанная точность), FlashAttention 2. Для получения кумулятивного положительного эффекта при совместном использовании этих методов необходимо выполнить ряд практических экспериментов. При настройке гиперпараметров обучения LLM сначала следует определить, какой размер пакета дает наилучшие результаты, а затем выбрать адекватные методы оптимизации используемых вычислительных ресурсов. Применение представленных методов позволит повысить эффективность использования вычислительных ресурсов при настройке больших языковых моделей и обеспечит сокращение необходимых для этого временных и финансовых затрат.*

Ключевые слова: *большая языковая модель, графический процессор, накопление градиента, смешанная точность, точная настройка LLM, Large Language Model, Low-Rank Adaptation*

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. A Survey of Large Language Models / W. Zhao [et al.] // ArXiv. 2023. Vol. 2303.18223. 124 p. DOI: 10.48550/arXiv.2303.18223
2. Down to Scale Up: A Guide to Parameter-Efficient Fine-Tuning / V. Lialin [et al.] // ArXiv. 2023. Vol. 2303.15647. 21 p. DOI: 10.48550/arXiv.2303.15647
3. Matrix Multiplication Background User's Guide // NVIDIA Documentation Hub. URL: http://docs.nvidia.com/deeplearning/performance/dl-performance-matrix-multiplication (accessed 26 Mar 2024).
4. Bekman S. Benchmarking Transformers with HF Trainer on a Single A100 40GB // Github. URL: http://github.com/huggingface/transformers/issues/15026 (accessed 26 Mar 2024).
5. LORA: Low-Rank Adaptation of Large Language Models / E. Hu [et al.] // ArXiv. 2021. Vol. 2106.09685. 26 p. DOI: 10.48550/arXiv.2106.09685
6. LLaMA-Adapter: Efficient Fine-Tuning of Language Models with Zero-Init Attention / R. Zhang [et al.] // ArXiv. 2023. Vol. 2303.16199. 22 p. DOI: 10.48550/arXiv.2303.16199
7. Delta Tuning: A Comprehensive Study of Parameter Efficient Methods for Pre-Trained Language Models / N. Ding [et al.] // ArXiv. 2022. Vol. 2203.06904. 49 p. DOI: 10.48550/arXiv.2203.06904
8. QA-LoRA: Quantization-Aware Low-Rank Adaptation of Large Language Models / Y. Xu [et al.] // ArXiv. 2023. Vol. 2309.14717. 16 p. DOI: 10.48550/arXiv.2309.14717
9. QDyLoRA: Quantized Dynamic Low-Rank Adaptation for Efficient Large Language Model Tuning / H. Rajabzadeh [et al.] // ArXiv. 2024. Vol. 2402.10462. 6 p. DOI: 10.48550/arXiv.2402.10462
10. Methods and Tools for Efficient Training on a Single GPU // Hugging Face Community. URL: http://huggingface.co/docs/transformers/perf_train_gpu_one (accessed 26 Mar 2024).
11. Goodfellow I., Bengio Y., Courville A. Optimization for Training Deep Model // Deep Learning. Cambridge (MA): MIT Press, 2016. Pp. 267–320.
12. Bekman S. Benchmarking Transformers with HF Trainer on RTX-309 // Github. URL: http://github.com/huggingface/transformers/issues/14608 (accessed 26 Mar 2024).
13. Linear/Fully Connected Layers User's Guide // NVIDIA Documentation Hub. URL: http://docs.nvidia.com/deeplearning/performance/dl-performance-fully-connected (accessed 26 Mar 2024).
14. Exploring Parameter-Efficient Fine-Tuning Techniques for Code Generation with Large Language Models / M. Weyssow [et al.] // ArXiv. 2023. Vol. 2308.10462. 23 p. DOI: 10.48550/arXiv.2308.10462
15. PTraining FP8 Large Language Models / H. Peng [et al.] // ArXiv. 2023. Vol. 2310.18313. 23 p. DOI: 10.48550/arXiv.2310.18313.10.48550/arXiv.2310.18313

Дата поступления: 18.07.2024
Решение о публикации: 13.08.2024