PETALS: Collaborative Inference and Fine-tuning of Large Models

Alexander Borzunov[*] HSE University, Yandex
Dmitry Baranchuk[*] Yandex
Tim Dettmers[*] University of Washington
Max Ryabinin[*] HSE University, Yandex
Younes Belkada[*] Hugging Face, ENS Paris-Saclay
Artem Chumachenko Yandex
Pavel Samygin Yandex School of Data Analysis
Colin Raffel Hugging Face

Abstract

Many NLP tasks benefit from using large language models (LLMs) that often have more than 100 billion parameters. With the release of BLOOM-176B and OPT-175B, everyone can download pretrained models of this scale. Still, using these models requires high-end hardware unavailable to many researchers. In some cases, LLMs can be used more affordably via RAM offloading or hosted APIs. However, these techniques have innate limitations: offloading is too slow for interactive inference, while APIs are not flexible enough for research that requires access to weights, attention or logits. In this work, we propose PETALS — a system for inference and fine-tuning of large models collaboratively by joining the resources of multiple parties. We demonstrate that this strategy outperforms offloading for very large models, running inference of BLOOM-176B on consumer GPUs with $\approx$ 1 step per second, which is enough for many interactive LLM applications. Unlike most inference APIs, PETALS also natively exposes hidden states of served models, allowing to train and share custom model extensions based on efficient fine-tuning methods. The system, its source code, and documentation are available at `https://petals.ml`.

1 Introduction

In recent years, the NLP community has found that pretrained language models can solve many practical tasks, through either fine-tuning (Radford et al., 2018) or simple prompting (Brown et al., 2020). Furthermore, performance tends to improve as scale increases (Radford et al., 2019; Kaplan et al., 2020). Following this trend, modern LLMs often have hundreds of billions of parameters (Brown et al., 2020; Rae et al., 2021; Zeng et al., 2021; Kim et al., 2021). Some of these LLMs were released publicly (Zhang et al., 2022; Khrushchev et al., 2022; Zeng et al., 2022). Most recently, the BigScience project has released BLOOM, a 176 billion parameter model supporting 46 natural and 13 programming languages (Scao et al., 2022).

While the public availability of 100B+ parameter models makes them easier to access, they remain difficult to use for the majority of researchers and practitioners due to memory and computational costs. For instance, OPT-175B and BLOOM-176B need over 350 GB accelerator memory for inference and significantly more for fine-tuning. As a result, these LLMs usually require multiple high-end GPUs or multi-node clusters to be run. Both of these options are extremely expensive, which limits research and potential applications of LLMs.

Several recent works aim to democratize LLMs by "offloading" model parameters to slower but cheaper memory (RAM or SSD), then running them on the accelerator layer by layer (Pudipeddi et al., 2020; Ren et al., 2021). This method allows running LLMs with a single low-end accelerator by loading parameters from RAM justin-time for each forward pass. Offloading can be efficient for processing many tokens in parallel, but it has inherently high latency: for example, generating one token at a time with BLOOM-176B takes at least *5.5 seconds* for the fastest RAM offloading setup and 22 seconds for the fastest SSD offloading. In addition, many computers do not have enough RAM to offload 175B parameters.

Another way to make LLMs more accessible is through public inference APIs, where one party hosts the model and lets others query it over the Internet (OpenAI; AI21; Forefront). Since most of the engineering work is done by the API owner, this is a relatively user-friendly option. However, APIs are often not flexible enough for research use: there is no way to change the model control flow or access internal states. On top of that, current API pricing can make some research projects prohibitively expensive (Liu et al., 2022a).

[*]Equal contribution. Correspondence to: `borzunov.alexander@gmail.com`

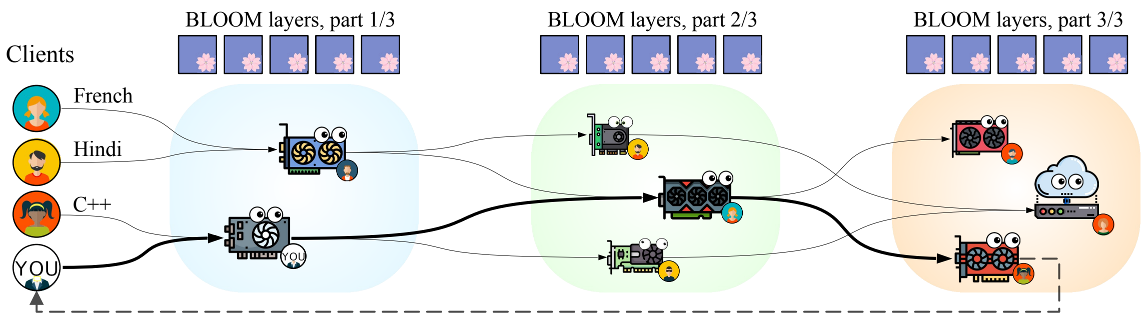


Figure 1: An overview of PETALS. Some participants (*clients*) want to use a pretrained language model to solve various tasks involving processing texts in natural (e.g., French, Hindi) or programming (e.g., C++) languages. They do it with help of other participants (*servers*), who hold various subsets of model layers on their GPUs. Each client chooses a sequence of servers so that it performs an inference or fine-tuning step in the least amount of time.

In this work, we explore an alternative strategy inspired by crowdsourced distributed training of neural networks from scratch (Ryabinin and Gusev, 2020). We introduce PETALS, a platform that allows multiple users to collaborate and perform inference and fine-tuning of large language models over the Internet. Each participant runs a server, a client or both. A *server* hosts a subset of model layers (typically, Transformer blocks) and handles requests from clients. A *client* can form a chain of pipeline-parallel consecutive servers to run the inference of the entire model (Section 2.1). Aside from inference, participants can fine-tune the model through parameter-efficient training methods like adapters (Houlsby et al., 2019) or prompt tuning (Lester et al., 2021) or by training entire layers (Section 2.2). Once trained, submodules can be shared on a model hub (Section 2.3), where others can use them for inference or further training. We demonstrate that existing 100B+ models can run efficiently in this setting with the help of several optimizations: dynamic quantization, prioritizing low-latency connections, and load balancing between servers (Section 3). Finally, we discuss limitations and possible future work (Appendix A).

2 Design and use cases

Practical usage of large language models can be broadly divided into two main scenarios: inference and parameter-efficient adaptation to downstream tasks. In this section, we outline the design of PETALS, showing how it handles both scenarios and also allows easily sharing trained adapters between the users of the system.

2.1 Inference of billion-scale models

When generating tokens, a client stores the model's token embeddings (which typically comprise a small fraction of the total parameter count and can fit in RAM in most modern laptops, servers, and workstations) locally and relies on servers to run Transformer blocks. Each server holds several *consecutive* blocks, the number of which depends on the server's available GPU memory. Before each inference session, the client finds a chain of servers that collectively hold all model layers.

Once the chain is formed, the client uses the local embedding layer to look up embedding vectors for prefix tokens, then sends those vectors to servers and receives new representations. Once the client obtains the outputs of the final block, it computes next token probabilities and repeats this process.

While the session is active, servers store attention keys and values from past client inputs and use them for subsequent inference steps. Clients also store past inputs to each server so that if any server fails or goes offline, another one can quickly take its place. The procedure for finding servers and recovering from failures is detailed in Section 3.2.

Client-side API. To generate tokens with PETALS, one first creates an *inference session*. An inference session iteratively takes inputs as PyTorch tensors, runs them through all Transformer blocks and returns final representations as PyTorch tensors. Under the hood, sessions form server chains, hold cache, and recover from server failures in a way that is transparent to the user. An example of using an inference session is shown in Figure 2.

System requirements. For BLOOM-176B inference, clients need at least 12 GB RAM, most of which is used to store 3.6B embedding parameters. We recommend at least 25 Mbit/s bidirectional bandwidth to avoid bottlenecks in network transfers. Simple greedy inference can use any CPU that runs PyTorch, but more advanced algorithms (e.g., beam search) may require a GPU.

```python
# Initialize distributed BLOOM model
model = DistributedBloomForCausalLM \
    .from_pretrained("bigscience/bloom-petals")
input_ids = tokenizer(prefix_text)

with model.inference_session() as session:
    # Session maintains a set of servers that
    # store attention KV from previous steps
    for _ in range(sequence_length):
        # Compute the word embeddings locally
        hid = model.word_embeddings(input_ids)
        # Run distributed Transformer blocks,
        # store attention KV for future steps
        hid = session.step(hid)
        # Sample the next token locally
        probs = model.lm_head(hid)
        input_ids = sample_next_token(probs)
```

Figure 2: A basic PyTorch code snippet for generation with a distributed BLOOM-176B model.

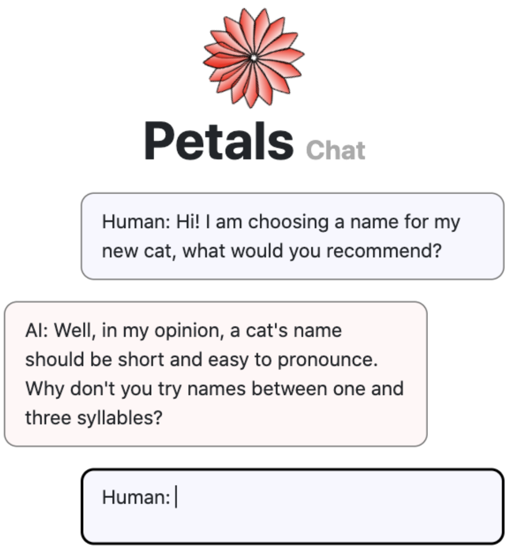
In turn, servers need at least 16 GB of CPU RAM, 100 Mbit/s bandwidth and a GPU with at least 8 GB of memory.

Chat application. We also provide an example application that lets users chat with LLMs in a messenger-like user interface (see Figure 3). The application supports BLOOM-176B and BLOOMZ-176B, a version of BLOOM fine-tuned to better perform in the zero-shot regime (Muennighoff et al., 2022). The application is comprised of the *frontend* and the *backend*. The frontend is a web page that allows users to communicate with the model by prompting it with text and receiving the generated output. The backend is a Flask web server that uses the PETALS client to run inference over the swarm. It accepts requests via HTTP or Websocket protocols, so anyone can develop their own applications using our backend for inference.

2.2 Training for downstream tasks

While LLMs achieve high quality on many problems with simple prompt engineering (Brown et al., 2020), they often need training to achieve the best results. Traditionally, this is done by fine-tuning all model parameters on the downstream task. However, for very large models, this strategy becomes impractical due to hardware requirements. For example, fine-tuning BLOOM-176B with Adam would require almost 3 TB of GPU memory to store model, gradients, and optimizer states.

To combat this issue, the NLP community has developed *parameter-efficient fine-tuning* methods that keep most of the pretrained model intact. Some of them (Sung et al., 2021; Guo et al., 2021) choose a subset of existing parameters, others (Hu et al., 2021; Houlsby et al., 2019; Liu et al., 2021b; Lester et al., 2021; Liu et al., 2021a, 2022a) augment the model with extra trainable weights.



Human: Hi! I am choosing a name for my new cat, what would you recommend?

AI: Well, in my opinion, a cat's name should be short and easy to pronounce. Why don't you try names between one and three syllables?

Human: |

Figure 3: A chat application that runs BLOOM-176B or BLOOMZ-176B over the PETALS swarm, available at `https://chat.petals.ml`

Despite their lower memory requirements, parameter-efficient approaches are often competitive with full model fine-tuning (Hu et al., 2021; Liu et al., 2021a; Yong and Nikoulina, 2022) and even outperform it in low-data regimes (Liu et al., 2022b). Another appealing property of these approaches for our use-case is that they allow rapidly switching a pretrained LLM between different uses.

Distributed fine-tuning. The core principle of fine-tuning in a distributed network is that clients "own" trained parameters while servers host original pretrained layers. Servers can run backpropagation through their layers and return gradients with respect to activations, but they *do not update the server-side parameters*. Thus, clients can simultaneously run different training tasks on the same set of servers without interfering with one another.

To illustrate this principle, we first review an example of soft prompt-tuning for text classification and then generalize it to other methods and tasks. Similarly to Section 2.1, clients store the embedding layers locally and rely on servers to compute the activations of Transformer blocks. In this fine-tuning scenario, a client needs to store trainable soft prompts (task-specific input embeddings) and a linear classification head.

For each training batch, the client routes its data through a chain of remote servers to compute sentence representations, then obtains predictions with the classifier head and computes the cross-entropy

```python
# Use distributed BLOOM with soft prompts
model = AutoModelForSequenceClassification \
    .from_pretrained(
        "bigscience/bloom-petals",
        tuning_mode="ptune", pre_seq_len=5)
# Define optimizer for prompts and linear head
opt = torch.optim.AdamW(model.parameters())

for input_ids, labels in data_loader:
    # Forward pass with local & remote layers
    out = model.forward(input_ids)
    loss = cross_entropy(out.logits, labels)

    # Distributed backward w.r.t. local params
    loss.backward() # Compute prompts.grad
    opt.step() # Update local params only
    opt.zero_grad()
```

Figure 4: A basic PyTorch code of soft prompt tuning for sequence classification with PETALS.

loss. During backpropagation, the client runs its data through the same chain of servers in reverse order to compute gradients for the learned prompt vectors. Having obtained those gradients, the client can use a regular PyTorch optimizer to update the parameters of both the head and the prompts, then proceed to the next minibatch.

User interface. To allow users greater flexibility in their training workloads, we made distributed backpropagation module compatible with the PyTorch Autograd engine. Like in the inference stage, this module handles fault tolerance and load balancing transparently to the user while allowing them to access intermediate activations and insert custom PyTorch modules. Figure 4 shows an example training code snippet.

This interface can also support other popular parameter-efficient fine-tuning algorithms, such as LoRA (Hu et al., 2021) or prefix tuning (Li and Liang, 2021). Finally, users can insert custom local modules after some of the existing blocks, which could allow use-cases like retrieval-augmented generation (Borgeaud et al., 2021; Lewis et al., 2020).

2.3 Sharing and reusing trained modules

Although most fine-tuned extensions for pretrained models can be easily shared as-is, simplifying the workflow for sharing these extensions enables users to more easily adapt the model to their target scenario. Indeed, existing model hubs (Wolf et al., 2020; TensorFlow Hub; PyTorch Hub) have gained immense popularity due to many supported models and ease of use, especially when vetting different pretrained models for a given problem. One particularly relevant project is AdapterHub (Pfeiffer et al., 2020), a repository of trained adapters accompanied by a library with implementations of different adaptation methods. While PETALS does not depend on AdapterHub, it is possible to leverage this library for training adapters in the distributed setting. Instead, we support sharing modules trained by users via the Hugging Face Hub (also used as a backend by AdapterHub). Its infrastructure and the corresponding open source library simplify the learning process for users already familiar with the ecosystem. Because the primary navigation mechanism on the Hugging Face Hub are tags that have been applied to uploaded modules, a user only needs to the task it was trained on and the model upon which the adapter was built. Uploading the weights and the code of the fine-tuned module is done by committing them to a Git repository. When navigating the Hub, users can choose the most suitable adapters by filtering the list of all available modules by the required tags.

3 Internal structure and optimizations

One of the primary considerations for distributed inference is its performance. It can be broken down into three main aspects: computation speed (5-year-old gaming GPU vs. new data center GPU), communication delay due to distance between nodes (intercontinental vs. local), and communication delay due to bandwidth (10 Mbit/s vs. 10 Gbit/s).

In terms of raw FLOPs, even consumer-grade GPUs like GeForce RTX 3070 could run a complete inference step of BLOOM-176B in less than a second (NVIDIA, 2020). However, the GPU memory can only hold a small fraction of model layers: running naïvely would require 44 RTX 3070 GPUs and 44 communication rounds. To make this more efficient, we use quantization to store more parameters per GPU, reducing the number of consecutive devices and communication rounds (Section 3.1). On top of that, each client prioritizes nearby servers to make communication rounds faster (Section 3.2).

3.1 Large model inference on consumer GPUs

We assume that each server has at least 16 GB of CPU RAM, 8 GB of GPU memory. From this assumption, one of the primary considerations is to reduce the model memory footprint, so that each device can hold more Transformer blocks.

For example, BLOOM has 176B parameters, which takes 352 GB of GPU memory in 16-bit precision. Thus, in the worst case, the model is distributed among 352 GB / 8 GB (per server)

Table 1: Zero-shot accuracy for OPT-175B and BLOOM-176B with 8-bit and 16-bit weights.

Model	Bits	HellaSwag	LAMBADA	WinoGrande	Avg
OPT-175B	16	78.5	74.7	72.6	75.3
	8	78.5	74.6	71.7	74.9
BLOOM	16	73.0	67.2	70.1	70.1
	8	72.8	68.1	70.1	70.3

Table 2: Generation throughput (tokens/s) for BLOOM-176B with 8-bit and 16-bit weights on 8× A100 GPUs.

Weights	Batch size		
	1	8	32
16-bit	4.18	31.3	100.6
8-bit	3.95	29.4	95.8

= 44 nodes. We can reduce both frequency and amount of data transfer in two ways. First, we can achieve this by compressing the hidden states exchanged between nodes. Second, we can compress the weights to 8-bit precision, reducing the number of nodes required to hold all layers. For BLOOM, this changes the number of required nodes from 44 to 22, which reduces latency in half and decreases the probability of a failure.

Compressing communication buffers. To send less data between subsequent pipeline stages, we use dynamic blockwise quantization (Dettmers et al., 2022b). We apply it to the hidden states before pipeline-parallel communication, as done in Ryabinin et al. (2023). Dynamic blockwise quantization halves the bandwidth requirements without any noticeable effect on generation quality.

Compressing model weights. We use 8-bit mixed matrix decomposition for matrix multiplication to quantize the weights to 8-bit precision and reduce the memory footprint compared to 16-bit weights, as suggested in (Dettmers et al., 2022a). This decomposition separates hidden states and weights into two portions: about 0.1% of 16-bit outlier and 99.9% of 8-bit regular values, which roughly halves the memory footprint.

As shown in Table 1, this method has little effect on LLM quality for major benchmarks. In terms of inference time, Table 2 demonstrates that quantization has about 5% of overhead with batch size 1 (20 tokens), but becomes negligible for larger batches.

3.2 Collaborating over the Internet

Another challenge is to provide *reliable* inference and training despite nodes joining, leaving or failing at any time. To address this, PETALS uses the `hivemind` library (Learning@home, 2020) for decentralized training with custom fault-tolerant algorithms for servers and clients detailed below.

Fault-tolerant generation. During inference, clients rely on servers to store attention keys and values for previous tokens. This introduces a potential problem if one or more servers disconnect (or fail) while generating a long sequence. To combat this, PETALS needs a way to recover from server failures transparently to the user.

A naive solution would be to restart the generation procedure, treating previously generated tokens as part of the prompt. This approach has two scaling issues. When generating longer sequences, the inference would have to restart more often, increasing the inference time superlinearly. Also, the more participants take part in the generation procedure, the higher the chance that one of them fails and the entire procedure needs to restart.

To reduce the time spent re-running computations Petals uses a special generation algorithm that supports partial restarts. To enable this, we make both clients and servers store previous activations. While each server stores past keys and values for its local blocks, each client remembers intermediate activations at every "junction" between servers (i.e., the activations it receives from the previous server and sends to the next one).

If one of the servers fail, the client only needs to replace the activations from that server. To do so, the client finds other servers holding the same blocks, then resends the cached activations that were sent to the previous (failed) server. Once this recovery is complete, the replacement server is in the same "inference state" as the rest of the chain, and the client can continue generating tokens.

Communication pattern. The algorithm above implies that clients send requests and receive responses from servers one by one, while servers do not directly pass activations to each other. This is suboptimal for sequential inference, where performance is bounded by the network latency.

To address this, we can make intermediate servers send the output activations both **(a)** directly to the next server and **(b)** back to the client. This way, the next server will start computations as soon as possible (after only one network hop instead of two hops), while the client will still be able to reuse the activations in case of server failures. Note that, in this case, sending two times more data does not worsen performance since, typically, sequential inference is not bounded by network bandwidth.

Server load balancing. First, we ensure that servers are distributed evenly among Transformer blocks. Formally, servers maximize the total model throughput by choosing the blocks with the lowest throughput, thus eliminating potential bottlenecks.

Here, the *block throughput* is the sum of throughputs of all servers hosting this block, while the *server throughput* is the minimum of its network and compute throughputs (in requests/sec), measured empirically before a server joins the system.

Each active server periodically announces its active blocks to a distributed hash table (Maymounkov and Mazieres, 2002). When a new server joins, it uses this information to choose an interval of blocks that contains blocks with the lowest throughput. The server only considers contiguous intervals, since hosting disjointed blocks would harm the inference latency. Once the server selects the best blocks to host, it reports them to the distributed hash table along with its own throughput.

Since peers may leave or fail at any time, all nodes periodically check if launching a rebalancing procedure would significantly improve the overall throughput. If it is the case, they switch layers until the throughput becomes near-optimal. In particular, if all peers serving certain blocks suddenly leave the system, this procedure quickly redistributes the remaining resources to close the emerged gaps.

Client-side routing. Next, we want clients to be able to find a sequence of servers that run the model in the least amount of time. During generation, clients process one or few tokens at a time; in practice, the inference time is mostly sensitive to the network latency. Thus, clients have to ping nearby servers to measure latency and then find the path with minimal time via beam search. Conversely, during fine-tuning one needs to process a batch of examples in parallel. Here, clients can split their batches between multiple servers using the algorithm from Ryabinin et al. (2023). If a server fails, a client removes it from consideration and reruns routing to find a replacement, possibly recovering inference caches as described above.

3.3 Benchmarks

We evaluate the performance of PETALS by running BLOOM-176B in emulated and real-world setups. Our first setup consists of 3 local servers, each running on an A100 80GB GPU. This is an optimistic scenario that requires the least amount of communication. In the second setup, we simulate 12 weaker devices by partitioning each A100-80GB into several virtual servers (3 large and 1 small). We evaluate the above setups with three network configurations: 1 Gbit/s with < 5 ms latency, 100 Mbit/s with < 5 ms latency and 100 Mbit/s with 100 ms latency[1]. The client nodes have 8 CPU cores and no GPU.

Next, we benchmark BLOOM in a real-world distributed setting with 14 smaller servers holding 2$\times$ RTX 3060, 4$\times$2080Ti, 2$\times$3090, 2$\times$A4000, and 4$\times$A5000 GPUs. These are personal servers and servers from university labs, spread across Europe and North America and connected to the Internet at speeds of 100–1000 Mbit/s. Four of the servers operate from under firewalls[2].

In Table 3, we report the performance of single-batch inference and parallel forward passes for batches of 128-token sequences. For inference, performance does not depend much on bandwidth or sequence length but degrades with higher latency. Parallel forward passes with large batches (used for fine-tuning and parallel inference) are affected by both bandwidth and latency.

Table 3: Performance of sequential inference steps and parallel forward passes. RTT is the round-trip latency.

Network	**Single-batch inference (steps/s)**		**Parallel forward (tokens/s)**	
	Sequence length		**Batch size**	
Bandwidth, RTT	128	2048	1	64
PETALS on 3 physical servers, with one A100 each				
1 Gbit/s, < 5 ms	1.71	1.54	70.0	253.6
100 Mbit/s, < 5 ms	1.66	1.49	56.4	182.0
100 Mbit/s, 100 ms	1.23	1.11	19.7	112.2
PETALS on 12 virtual servers				
1 Gbit/s, < 5 ms	1.24	1.06	37.9	180.0
100 Mbit/s, < 5 ms	1.24	1.05	25.6	66.6
100 Mbit/s, 100 ms	0.57	0.53	5.8	44.3
PETALS on 14 real servers in Europe and North America				
Real world	0.83	0.79	32.6	179.4
Offloading, max. speed on 1x A100				
256 Gbit/s	0.18	0.18	2.7	170.3
128 Gbit/s	0.09	0.09	2.4	152.8
Offloading, max. speed on 3x A100				
256 Gbit/s	0.09	0.09	5.1	325.1
128 Gbit/s	0.05	0.05	3.5	226.3

[1]We simulate network conditions using `tc qdisc`.

[2]We use the Circuit Relay protocol (libp2p, 2022) to traverse NATs and firewalls.

We also test the effect of having multiple clients. For 12 servers with 100 Mbit/s bandwidth and 100 ms latency, if 8 clients run inference concurrently, each of them gets $\approx 20\%$ slowdown compared to the case when it runs inference alone.

Additionally, we compare PETALS with parameter offloading to run large models with limited resources (Ren et al., 2021; Rajbhandari et al., 2021). For the offloading benchmark we calculate the maximum inference and forward training throughput to receive an upper bound on offloading performance. We base our offloading numbers on the best possible hardware setup for offloading: CPU RAM offloading via PCIe 4.0 with 16 PCIe lanes per GPU and PCIe switches for pairs of GPUs.

We calculate the maximum throughput for offloading as follows. In 8-bit, the model uses 1 GB of memory per billion parameters while PCIe 4.0 with 16 lanes has a throughput of 256 Gbit/s (or 128 Gbit/s if two GPUs are behind a PCIe switch). As such, offloading 176B parameters takes 5.5 seconds for a regular setup and 11 seconds for a multi-GPU setup. We assume an offloading latency of zero for the upper bound estimation.

These results are also shown in Table 3. We can see that offloading is about an order of magnitude slower for single-batch inference compared to PETALS. For the fine-tuning forward pass, offloading is competitive if multiple GPUs are used and the networking for PETALS is limited to 100 Mbit/s or has high latency. In other cases, PETALS offers higher throughput than offloading for training.

4 Conclusion

This paper introduces PETALS, a system for efficient collaborative inference and fine-tuning of large language models. We offer a user-friendly generation interface and a flexible API to access models served over the Internet. We use 8-bit compression that reduces the resource requirements to run very large models. In addition, we develop algorithms for reliable routing and load balancing.

With the release of this system, we hope to broaden access to LLMs and pave the road to applications, studies or research questions that were previously not possible or simply too expensive.

Running LLMs over the Internet raises a broad range of related questions. One of them is privacy: how to avoid revealing private data to outside peers. Another challenge is to ensure that participants can benefit from this system equitably, i.e. in proportion to their contribution. We discuss future problems such as privacy, security, and incentive structures in Appendix A.

Limitations

An important limitation of our work is *data privacy*: the intermediate activations of the model for given inputs are sent to the servers without any encryption. As such, it might be possible for people hosting the servers to recover the user's input data. Another limitation is *security*: while there are ways to detect and penalize peers sending faulty outputs, still there is a chance that peers may do that due to faulty hardware or a malicious intent.

Thus, we recommend users working with sensitive data to only use servers hosted by institutions trusted to process this data or set up an isolated PETALS swarm.

We discuss these limitations in more detail in Appendix A and acknowledge that the development of methods for privacy-preserving and secure decentralized inference without performance penalties remains an open research problem.

Ethics Statement

This work introduces a general-purpose algorithm for decentralized inference of large models, aiming to simplify access to the latest research in deep learning. Thus, we do not envision any direct negative impacts from our research aside from granting the broader public an ability to interact with LLMs trained on uncurated web-crawled data. However, all models we serve are already in open access and thus can be exposed via APIs or other means.

Acknowledgements

The authors thank Zheng-Xin Yong, Ilya Dimov, Yozh, Teven Le Scao, Stas Bekman, and Haokun Liu for helpful discussions. We also thank Teven Le Scao for his help in designing Figure 1. A part of the experiments was conducted on a personal server of Elena Voita.

References

AI21. Jurassic-1 language models. "https://studio.ai21.com/docs/jurassic1-language-models". Accessed: 2022-06-22.

Sid Black, Stella Biderman, Eric Hallahan, Quentin Anthony, Leo Gao, Laurence Golding, Horace He, Connor Leahy, Kyle McDonell, Jason Phang, Michael Pieler, USVSN Sai Prashanth, Shivanshu Purohit, Laria Reynolds, Jonathan Tow, Ben Wang, and Samuel Weinbach. 2022. GPT-NeoX-20B: An open-source autoregressive language model.

Sebastian Borgeaud, Arthur Mensch, Jordan Hoffmann, Trevor Cai, Eliza Rutherford, Katie Millican, George van den Driessche, Jean-Baptiste Lespiau, Bogdan Damoc, Aidan Clark, Diego de Las Casas, Aurelia Guy, Jacob Menick, Roman Ring, Tom Hennigan, Saffron Huang, Loren Maggiore, Chris Jones, Albin Cassirer, Andy Brock, Michela Paganini, Geoffrey Irving, Oriol Vinyals, Simon Osindero, Karen Simonyan, Jack W. Rae, Erich Elsen, and Laurent Sifre. 2021. Improving language models by retrieving from trillions of tokens.

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

Tim Dettmers, Mike Lewis, Younes Belkada, and Luke Zettlemoyer. 2022a. LLM.int8(): 8-bit matrix multiplication for transformers at scale. *ArXiv*, abs/2208.07339.

Tim Dettmers, Mike Lewis, Sam Shleifer, and Luke Zettlemoyer. 2022b. 8-bit optimizers via block-wise quantization. *International Conference on Learning Representations (ICLR)*.

Nan Du, Yanping Huang, Andrew M. Dai, Simon Tong, Dmitry Lepikhin, Yuanzhong Xu, Maxim Krikun, Yanqi Zhou, Adams Wei Yu, Orhan Firat, Barret Zoph, Liam Fedus, Maarten Bosma, Zongwei Zhou, Tao Wang, Yu Emma Wang, Kellie Webster, Marie Pellat, Kevin Robinson, Kathy Meier-Hellstern, Toju Duke, Lucas Dixon, Kun Zhang, Quoc V. Le, Yonghui Wu, Zhifeng Chen, and Claire Cui. 2021. GLaM: efficient scaling of language models with mixture-of-experts. *CoRR*, abs/2112.06905.

David Evans, Vladimir Kolesnikov, Mike Rosulek, et al. 2018. A pragmatic introduction to secure multi-party computation. *Foundations and Trends in Privacy and Security*, 2(2-3):70–246.

Hugging Face and contributors. 2020. Accelerate: Run your raw PyTorch training script on any kind of device. *GitHub. Note: https://github.com/huggingface/datasets*, 1.

William Fedus, Barret Zoph, and Noam Shazeer. 2021. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity.

Forefront. Powerful language models a click away. "https://www.forefront.ai/". Accessed: 2022-06-22.

Leo Gao, Jonathan Tow, Stella Biderman, Sid Black, Anthony DiPofi, Charles Foster, Laurence Golding, Jeffrey Hsu, Kyle McDonell, Niklas Muennighoff, Jason Phang, Laria Reynolds, Eric Tang, Anish Thite, Ben Wang, Kevin Wang, and Andy Zou. 2021. A framework for few-shot language model evaluation.

Sebastian Gehrmann, Abhik Bhattacharjee, Abinaya Mahendiran, Alex Wang, Alexandros Papangelis, Aman Madaan, Angelina McMillan-Major, Anna Shvets, Ashish Upadhyay, Bingsheng Yao, Bryan Wilie, Chandra Bhagavatula, Chaobin You, Craig Thomson, Cristina Garbacea, Dakuo Wang, Daniel Deutsch, Deyi Xiong, Di Jin, Dimitra Gkatzia, Dragomir Radev, Elizabeth Clark, Esin Durmus, Faisal Ladhak, Filip Ginter, Genta Indra Winata, Hendrik Strobelt, Hiroaki Hayashi, Jekaterina Novikova, Jenna Kanerva, Jenny Chim, Jiawei Zhou, Jordan Clive, Joshua Maynez, João Sedoc, Juraj Juraska, Kaustubh Dhole, Khyathi Raghavi Chandu, Laura Perez-Beltrachini, Leonardo F. R. Ribeiro, Lewis Tunstall, Li Zhang, Mahima Pushkarna, Mathias Creutz, Michael White, Mihir Sanjay Kale, Moussa Kamal Eddine, Nico Daheim, Nishant Subramani, Ondrej Dusek, Paul Pu Liang, Pawan Sasanka Ammanamanchi, Qi Zhu, Ratish Puduppully, Reno Kriz, Rifat Shahriyar, Ronald Cardenas, Saad Mahamood, Salomey Osei, Samuel Cahyawijaya, Sanja Štajner, Sebastien Montella, Shailza, Shailza Jolly, Simon Mille, Tahmid Hasan, Tianhao Shen, Tosin Adewumi, Vikas Raunak, Vipul Raheja, Vitaly Nikolaev, Vivian Tsai, Yacine Jernite, Ying Xu, Yisi Sang, Yixin Liu, and Yufang Hou. 2022. GEMv2: multilingual NLG benchmarking in a single line of code.

Demi Guo, Alexander M Rush, and Yoon Kim. 2021. Parameter-efficient transfer learning with diff pruning. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics*.

Neil Houlsby, Andrei Giurgiu, Stanislaw Jastrzebski, Bruna Morrone, Quentin De Laroussilhe, Andrea Gesmundo, Mona Attariyan, and Sylvain Gelly. 2019. Parameter-efficient transfer learning for NLP. In *International Conference on Machine Learning*, pages 2790–2799. PMLR.

Edward Hu, Yelong Shen, Phil Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Lu Wang, and Weizhu Chen. 2021. LoRA: low-rank adaptation of large language models.

Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. 2020. Scaling laws for neural language models.

Michael Khrushchev, Ruslan Vasilev, Nikolay Zinov, Alexey Petrov, and Yandex. 2022. YaLM 100B. https://huggingface.co/yandex/yalm-100b.

Douwe Kiela, Max Bartolo, Yixin Nie, Divyansh Kaushik, Atticus Geiger, Zhengxuan Wu, Bertie Vidgen, Grusha Prasad, Amanpreet Singh, Pratik Ringshia, Zhiyi Ma, Tristan Thrush, Sebastian Riedel, Zeerak Waseem, Pontus Stenetorp, Robin Jia, Mohit Bansal, Christopher Potts, and Adina Williams. 2021. Dynabench: Rethinking benchmarking in NLP.

Boseop Kim, HyoungSeok Kim, Sang-Woo Lee, Gichang Lee, Dong-Hyun Kwak, Dong Hyeon Jeon, Sunghyun Park, Sungju Kim, Seonhoon Kim, Dongpil Seo, Heungsub Lee, Minyoung Jeong, Sungjae Lee, Minsub Kim, SukHyun Ko, Seokhun Kim, Taeyong Park, Jinuk Kim, Soyoung Kang, Na-Hyeon Ryu, Kang Min Yoo, Minsuk Chang, Soobin Suh, Sookyo In, Jinseong Park, Kyungduk Kim, Hiun Kim, Jisu Jeong, Yong Goo Yeo, Donghoon Ham, Dongju Park, Min Young Lee, Jaewook Kang, Inho Kang, Jung-Woo Ha, Woo-Myoung Park, and Nako Sung. 2021. What changes can large-scale language models bring? intensive study on hyperclova: Billions-scale korean generative pretrained transformers. *CoRR*, abs/2109.04650.

Team Learning@home. 2020. Hivemind: a Library for Decentralized Deep Learning. `https://github.com/learning-at-home/hivemind`.

Dmitry Lepikhin, H. Lee, Yuanzhong Xu, Dehao Chen, Orhan Firat, Y. Huang, M. Krikun, Noam Shazeer, and Z. Chen. 2020. GShard: scaling giant models with conditional computation and automatic sharding. *ArXiv*, abs/2006.16668.

Brian Lester, Rami Al-Rfou, and Noah Constant. 2021. The power of scale for parameter-efficient prompt tuning. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 3045–3059, Online and Punta Cana, Dominican Republic. Association for Computational Linguistics.

Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. 2020. Retrieval-augmented generation for knowledge-intensive nlp tasks. In *Advances in Neural Information Processing Systems*, volume 33, pages 9459–9474. Curran Associates, Inc.

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.

libp2p. 2022. libp2p circuit relay. `https://docs.libp2p.io/concepts/nat/circuit-relay/`.

Haokun Liu, Derek Tam, Mohammed Muqeeth, Jay Mohta, Tenghao Huang, Mohit Bansal, and Colin Raffel. 2022a. Few-shot parameter-efficient fine-tuning is better and cheaper than in-context learning.

Haokun Liu, Derek Tam, Mohammed Muqeeth, Jay Mohta, Tenghao Huang, Mohit Bansal, and Colin Raffel. 2022b. Few-shot parameter-efficient fine-tuning is better and cheaper than in-context learning.

Xiao Liu, Kaixuan Ji, Yicheng Fu, Zhengxiao Du, Zhilin Yang, and Jie Tang. 2021a. P-tuning v2: Prompt tuning can be comparable to fine-tuning universally across scales and tasks. *arXiv preprint arXiv:2110.07602*.

Xiao Liu, Yanan Zheng, Zhengxiao Du, Ming Ding, Yujie Qian, Zhilin Yang, and Jie Tang. 2021b. GPT understands, too. *arXiv:2103.10385*.

Petar Maymounkov and David Mazieres. 2002. Kademlia: A peer-to-peer information system based on the xor metric. In *International Workshop on Peer-to-Peer Systems*, pages 53–65. Springer.

Niklas Muennighoff, Thomas Wang, Lintang Sutawika, Adam Roberts, Stella Biderman, Teven Le Scao, M Saiful Bari, Sheng Shen, Zheng-Xin Yong, Hailey Schoelkopf, et al. 2022. Crosslingual generalization through multitask finetuning. *arXiv preprint arXiv:2211.01786*.

Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, et al. 2021. Efficient large-scale language model training on gpu clusters. *arXiv preprint arXiv:2104.04473*.

NVIDIA. 2020. NVIDIA Ampere GA102 GPU architecture.

NVIDIA. 2022. NVIDIA confidential computing. `https://www.nvidia.com/en-in/data-center/solutions/confidential-computing/`.

OpenAI. Build next-gen apps with OpenAI's powerful models. https://openai.com/api. Accessed: 2022-06-22.

Jonas Pfeiffer, Andreas Rücklé, Clifton Poth, Aishwarya Kamath, Ivan Vulić, Sebastian Ruder, Kyunghyun Cho, and Iryna Gurevych. 2020. Adapterhub: A framework for adapting transformers. *arXiv preprint arXiv:2007.07779*.

Bharadwaj Pudipeddi, Maral Mesmakhosroshahi, Jinwen Xi, and Sujeeth Bharadwaj. 2020. Training large neural networks with constant memory using a new execution algorithm. *arXiv preprint arXiv:2002.05645*.

PyTorch Hub. PyTorch Hub. `https://pytorch.org/hub/`. Accessed: 2021-10-04.

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

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

Jack W Rae, Sebastian Borgeaud, Trevor Cai, Katie Millican, Jordan Hoffmann, Francis Song, John Aslanides, Sarah Henderson, Roman Ring, Susannah Young, et al. 2021. Scaling language models: Methods, analysis & insights from training gopher. *arXiv preprint arXiv:2112.11446*.

Colin Raffel. 2021. A call to build models like we build open-source software.

Samyam Rajbhandari, Olatunji Ruwase, Jeff Rasley, Shaden Smith, and Yuxiong He. 2021. Zero-infinity: Breaking the gpu memory wall for extreme scale deep learning. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–14.

Jie Ren, Samyam Rajbhandari, Reza Yazdani Aminabadi, Olatunji Ruwase, Shuangyan Yang, Minjia Zhang, Dong Li, and Yuxiong He. 2021. Zero-offload: Democratizing billion-scale model training.

Max Ryabinin, Tim Dettmers, Michael Diskin, and Alexander Borzunov. 2023. SWARM parallelism: Training large models can be surprisingly communication-efficient. *arXiv preprint arXiv:2301.11913*.

Max Ryabinin and Anton Gusev. 2020. Towards crowdsourced training of large neural networks using decentralized mixture-of-experts. In *Advances in Neural Information Processing Systems*, volume 33, pages 3659–3672. Curran Associates, Inc.

Teven Le Scao, Angela Fan, Christopher Akiki, Ellie Pavlick, Suzana Ilić, Daniel Hesslow, Roman Castagné, Alexandra Sasha Luccioni, François Yvon, Matthias Gallé, et al. 2022. BLOOM: a 176B-parameter open-access multilingual language model. *arXiv preprint arXiv:2211.05100*.

Yi-Lin Sung, Varun Nair, and Colin Raffel. 2021. Training neural networks with fixed sparse masks. *Advances in Neural Information Processing Systems*.

TensorFlow Hub. TensorFlow Hub. `https://www.tensorflow.org/hub`. Accessed: 2021-10-04.

Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. 2023. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*.

Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Remi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander Rush. 2020. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 38–45, Online. Association for Computational Linguistics.

Zheng-Xin Yong and Vassilina Nikoulina. 2022. Adapting bigscience multilingual model to unseen languages.

Aohan Zeng, Xiao Liu, Zhengxiao Du, Ming Ding, Qinkai Zheng, Hanyu Lai, Zihan Wang, Zhuoyi Yang, Jifan Yu, Xiaohan Zhang, Wendi Zheng, Xiao Xia, Yifan Xu, Weng Lam Tam, Yuxiao Dong, Zixuan Ma, Jiaao He, Zhenbo Sun, Jidong Zhai, Wenguang Chen, Guoyang Zeng, Xu Han, Weilin Zhao, Zhiyuan Liu, Yufei Xue, Shan Wang, Jiecai Shan, Haohan Jiang, Zhengang Guo, Peng Zhang, and Jie Tang. 2022. GLM-130B: An open bilingual pre-trained model.

Wei Zeng, Xiaozhe Ren, Teng Su, Hui Wang, Yi Liao, Zhiwei Wang, Xin Jiang, ZhenZhang Yang, Kaisheng Wang, Xiaoda Zhang, Chen Li, Ziyan Gong, Yifan Yao, Xinjing Huang, Jun Wang, Jianfeng Yu, Qi Guo, Yue Yu, Yan Zhang, Jin Wang, Hengtao Tao, Dasen Yan, Zexuan Yi, Fang Peng, Fangqing Jiang, Han Zhang, Lingfeng Deng, Yehong Zhang, Zhe Lin, Chao Zhang, Shaojie Zhang, Mingyue Guo, Shanzhi Gu, Gaojun Fan, Yaowei Wang, Xuefeng Jin, Qun Liu, and Yonghong Tian. 2021. Pangu-α: Large-scale autoregressive pretrained chinese language models with auto-parallel computation. *CoRR*, abs/2104.12369.

Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, Todor Mihaylov, Myle Ott, Sam Shleifer, Kurt Shuster, Daniel Simig, Punit Singh Koura, Anjali Sridhar, Tianlu Wang, and Luke Zettlemoyer. 2022. OPT: open pre-trained transformer language models.

Appendix

A Discussion and future work

Incentives for peers to contribute. In PETALS, peers using the client are not required to run a server. This may lead to an imbalance between supply (peers who dedicate GPUs to serve model layers) and demand (peers using the servers to perform inference or fine-tuning for their own needs) in the network. One way to encourage users to serve model layers is to introduce a system of *incentives*: peers running servers would earn special *points*, which can be spent on high-priority inference and fine-tuning or exchanged for other rewards.

Privacy. A key limitation of our approach is that peers serving the first layers of the model can use their inputs to recover input tokens. Thus, clients working with sensitive data should only use the servers hosted by institutions trusted to process this data. This can be achieved with the `allowed_servers` parameter that limits the set of servers a client can use. Alternatively, users can set up their own isolated Petals swarm.

This limitation may be addressed in future work, leveraging the fields of secure multi-party computing (Evans et al., 2018) or privacy-preserving hardware (NVIDIA, 2022).

Security. We assume that servers in our system are run by many independent parties. In practice, some of them may turn out to be faulty and return incorrect outputs instead of the actual results of forward and backward passes. This may happen due to a malicious intent to influence other people's outputs or, when rewards are introduced (as described above), to earn a reward for serving layers without actually performing the calculations.

A possible way to address these issues would be to use an economically motivated approach. Some servers may vouch for the correctness of their outputs (e.g., in exchange for increased inference price) by depositing a certain number of points as a pledge. Then, for each request, they announce a cryptographic hash of the input and output tensors, so anyone having the inputs can check whether the outputs are correct.

If someone finds a mismatch confirmed by a trusted third party, they can claim the server's pledge as a reward. In practice, it may be a client who suspects that they received wrong outputs or a "bounty hunter" sending requests to different servers in the hope of catching errors. While this approach still leaves a chance of receiving wrong outputs, it makes cheating costly and creates an incentive to quickly expose the malicious servers.

Making changes to the main model. As discussed in Section 2.2, distributed parameter-efficient fine-tuning makes it easy for users to apply the base model to new tasks. In Section 2.3, we also described how these updates can be easily shared and reused by others. This capability provides a meaningful step towards *collaborative* improvement of machine learning models (Raffel, 2021): as more and more users train the base model, it will effectively become more capable over time.

Furthermore, we might expect the model parameters that perform best on a specific task to change over time. Similarly to version control systems for code, it would be useful to track versions of fine-tuned model parameters as they change. A system for rapidly testing the performance of a set of parameters on "living benchmarks" (Kiela et al., 2021; Gehrmann et al., 2022; Gao et al., 2021) would be valuable to ensure that subsequent versions improve the desired capabilities.

Apart from adaptation to new tasks, it would also be useful to eventually update the main model. Ideally, such updates could be tracked in a principled way. Users of PETALS could specify the versions of the model they want to use, and servers could indicate which versions they support. Introducing a newer version of the model then reduces to adding a new group of layers, which then naturally supersedes older parameters based on the approach from Section 3.2. Similarly, fine-tuned adapters could be annotated with tags denoting the model version they are applicable for. Such fine-grained model versioning is currently uncommon but would be straightforward to add to PETALS.