**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 significantly outperforms offloading for very large models, running inference of BLOOM-176B on consumer GPUs with \( \approx 1 \) step per second. Unlike most inference APIs, PETALS also natively exposes the hidden states of served models, allowing its users to train and share custom model extensions based on efficient fine-tuning methods.

### 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 language models often have hundreds of billions of parameters [Brown et al., 2020; Rae et al., 2021; Zeng et al., 2021; Kim et al., 2021]. Several research groups released pretrained LLMs with over 100B parameters [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 350GB 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 the potential research directions and applications of large language models.

*Equal contribution. Correspondence to: petals-dev@googlegroups.com

1PETALS source code and documentation are available at [https://petals.ml](https://petals.ml)

Workshop on Broadening Research Collaborations in ML (NeurIPS 2022).
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 just in 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).

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 holds a subset of model layers (typically, Transformer blocks) on its local device 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.

In Section 3, 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. Finally, in Section 4, we discuss incentives for participating in the system, security and privacy, and how the model can be updated over time.

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
# Initialize distributed BLOOM model
model = AutoModelForCausalLM.from_pretrained("bigscience/distributed-bloom")
input_ids = tokenizer(prefix_text)

with model.inference_session() as session:
    for _ in range(sequence_length):
        hidden = model.word_embeddings(input_ids)
        hidden = session.step(hidden)
        probs = model.lm_head(hidden)
        input_ids = sample_next_token(probs)

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

**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.

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

**Graphical user interface.** We also provide an example application that lets a user chat with the model in a messenger-like user interface. The interface is divided into two main blocks: the *frontend* and the *backend* application. We use Hugging Face Spaces as the backend application that runs greedy inference on the CPU for each request. It is easily usable via Python’s `requests` library; therefore, anyone can use this setup as a backend and build their own frontend applications in any format. To give a better idea of how to use this backend, we provide an example frontend application using Hugging Face Spaces and the `streamlit` API. In this browser application, users can communicate with the model by prompting it with text and receiving the generated output.

### 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 extremely 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...
Initialize distributed BLOOM with soft prompts
model = AutoModelForPromptTuning.from_pretrained("bigscience/distributed-bloom")
# Define optimizer for prompts and linear head
optimizer = torch.optim.AdamW(model.parameters())

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

    # Distributed backward w.r.t. local params
    loss.backward()  # Compute model.prompts.grad
    optimizer.step()  # Update local params only
    optimizer.zero_grad()  # Update remote components

Figure 3: A basic PyTorch code of soft prompt tuning for sequence classification with PETALS.

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.

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 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 3 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 \cite{NVIDIA}. However, the GPU memory can only hold a small fraction of model layers: running naively 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) = 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 \cite{Dettmers2022}. We apply it to the hidden states before pipeline-parallel communication, as done in \cite{Ryabinin2021}. 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 \cite{Dettmers2022}. 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 important 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 \cite{Learning@home} for decentralized training and custom fault-tolerant protocols for servers and clients.

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

<table>
<thead>
<tr>
<th>Model</th>
<th>Bits</th>
<th>HellaSwag</th>
<th>LAMBADA</th>
<th>WinoGrande</th>
<th>Avg</th>
</tr>
</thead>
<tbody>
<tr>
<td>OPT-175B</td>
<td>16</td>
<td>78.5</td>
<td>74.7</td>
<td>72.6</td>
<td>75.3</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>78.5</td>
<td>74.6</td>
<td>71.7</td>
<td>74.9</td>
</tr>
<tr>
<td>BLOOM</td>
<td>16</td>
<td>73.0</td>
<td>67.2</td>
<td>70.1</td>
<td>70.1</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>72.8</td>
<td>68.1</td>
<td>70.1</td>
<td>70.3</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Weights</th>
<th>Batch size</th>
<th>1</th>
<th>8</th>
<th>32</th>
</tr>
</thead>
<tbody>
<tr>
<td>16-bit</td>
<td></td>
<td>4.18</td>
<td>31.3</td>
<td>100.6</td>
</tr>
<tr>
<td>8-bit</td>
<td></td>
<td>3.95</td>
<td>29.4</td>
<td>95.8</td>
</tr>
</tbody>
</table>

Each 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 identify an interval of blocks that contains most blocks with the worst throughput. This interval is always contiguous, since splitting it would harm the inference latency. Once the server has selected its layers, it measures its own throughput (both network and compute) and announces it to the distributed hash table.

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. (2021). If a server fails during training or inference, a client removes it from consideration and reruns routing to find a replacement. During inference, the client sends all previous inputs to the replacement server, so that it has the same attention keys and values.

### 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. The client-side nodes have 8 CPU cores and no GPU.

Next, we benchmark BLOOM in a real-world distributed setting with 14 smaller servers holding RTX 2×3060, 4×2080Ti, 2×3090, 2×A4000, and 4×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.

In Table 3, we report the performance of sequential inference and training-time forward passes. For inference, performance does not depend much on bandwidth or sequence length but degrades in

---

2 We simulate network conditions with [https://github.com/magnific0/wondershaper](https://github.com/magnific0/wondershaper) which uses tc qdisc

3 We use the Circuit Relay protocol from libp2p to traverse NATs and firewalls, see [https://docs.libp2p.io/concepts/circuit-relay/](https://docs.libp2p.io/concepts/circuit-relay/)
Table 3: Performance of sequential inference steps and training-time forward passes.

<table>
<thead>
<tr>
<th>Network</th>
<th>Inference (steps/s)</th>
<th>Training-time forward (tokens/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Sequence length</td>
<td>Batch size</td>
</tr>
<tr>
<td></td>
<td>128</td>
<td>2048</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>64</td>
</tr>
<tr>
<td>Offloading, max. speed on 1x A100</td>
<td></td>
<td></td>
</tr>
<tr>
<td>256 Gbit/s</td>
<td>0.18</td>
<td>0.18</td>
</tr>
<tr>
<td>128 Gbit/s</td>
<td>0.09</td>
<td>0.09</td>
</tr>
<tr>
<td>Offloading, max. speed on 3x A100</td>
<td></td>
<td></td>
</tr>
<tr>
<td>256 Gbit/s</td>
<td>0.09</td>
<td>0.09</td>
</tr>
<tr>
<td>128 Gbit/s</td>
<td>0.05</td>
<td>0.05</td>
</tr>
<tr>
<td>PETALS on 3 physical servers, with one A100 each</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 Gbit/s</td>
<td>&lt; 5 ms</td>
<td>1.22</td>
</tr>
<tr>
<td>100 Mbit/s</td>
<td>&lt; 5 ms</td>
<td>1.19</td>
</tr>
<tr>
<td>100 Mbit/s</td>
<td>100 ms</td>
<td>0.89</td>
</tr>
<tr>
<td>PETALS on 12 virtual servers, simulated on 3x A100</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 Gbit/s</td>
<td>&lt; 5 ms</td>
<td>0.97</td>
</tr>
<tr>
<td>100 Mbit/s</td>
<td>&lt; 5 ms</td>
<td>0.97</td>
</tr>
<tr>
<td>100 Mbit/s</td>
<td>100 ms</td>
<td>0.44</td>
</tr>
<tr>
<td>PETALS on 14 real servers in Europe and North America</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Real world</td>
<td>0.68</td>
<td>0.61</td>
</tr>
</tbody>
</table>

high-latency settings, especially for 12 virtual servers. In turn, training-time forward passes for large batches are affected by both bandwidth and latency.

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 ≈ 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 inference compared to PETALS. For the training-time 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 Discussion and future work

Incentives for peers to contribute. In PETALS, peers using the client are not required to run a server. Naturally, 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 would be 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.
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.

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 trusted institutions that are allowed to process this data. This limitation may be addressed in future work using secure multi-party computing (Evans et al., 2018) or privacy-preserving hardware (NVIDIA, 2022).

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.

5 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.

Since PETALS is open-source, we would like it to evolve based on the community’s feedback, incorporating relevant research advances and adding support for features in demand. With the release of this system, we hope to broaden access to large language models and pave the road to applications, studies or research questions that were previously not possible or simply too expensive.

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


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.


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

12