DISTRIBUTED INFERENCE AND FINE-TUNING OF LARGE LANGUAGE MODELS OVER THE INTERNET

Anonymous authors
Paper under double-blind review

ABSTRACT

Large language models (LLMs) are useful in many NLP tasks and become more capable with size, scaling to over 100 billion parameters. With the release of BLOOM-176B and OPT-175B, everyone can download pretrained models of this scale. Still, using a pre-trained 100B+ model requires high-end hardware, making it inaccessible to most researchers. Recent studies in memory-efficient training (e.g. offloading) could alleviate these costs, but they do not cover important use cases of LLMs, such as autoregressive inference. In this work, we investigate methods for cost-efficient inference of large language models, comparing local and distributed strategies. We observe that a *large enough* model (100B+) could run efficiently on geodistributed devices in a consumer-grade network, for example by connecting existing compute resources of multiple research groups or pooling under-utilized compute from multiple cloud regions. To run LLMs in this unconventional setting, we develop a fault-tolerant algorithm for inferencing language models. We propose PETALS – a decentralized system for running LLMs – and show that it can run BLOOM-176B over the Internet over $10\times$ faster than offloading for sequential generation. We evaluate the performance of our system in both simulated conditions and an actual distributed system spanning two continents. The design of PETALS allows participants to inference, and fine-tune, or inference fine-tuned models simultaneously without affecting each other's results.

1 INTRODUCTION

In recent years, the NLP community has found that pretrained language models greatly accelerated progress on many research problems through either fine-tuning (Radford et al., 2018) or simple prompting (Brown et al., 2020). Furthermore, their quality tends to improve as we increase model scale (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). Most recently, several research groups open-sources their pretrained LLMs with over 100B parameters for everyone to use (Zhang et al., 2022; Khrushchev et al., 2022; Zeng et al., 2022).

Even though these models are publicly available, they are still difficult use due to the sheer size in terms of parameters. For example, OPT-175B and BLOOM-176B need over 350GB accelerator memory for inference and significantly more for fine-tuning. As a result, even basic inference for large language models requires multiple high-end GPUs or multi-node clusters. Several recent studies propose algorithms for running large models with more affordable hardware (Pudipeddi et al., 2020; Ren et al., 2021), e.g. by offloading parameters to RAM. However, these techniques are inefficient in many practical LLM usage scenarios as we show in Section 3.1.

In this work, we search for a cost-effective way of running large pre-trained language models in their main use cases: inference, in-context learning, fine-tuning.We systematically analyze latency and throughput for training and inference tasks for these use cases and determine which factors become dominant for very large models. Notably, for models with over 100B parameters, communicating activations over a slow network can be faster than swapping layers from local RAM or SSD. Based on these observations, it should be possible to run LLMs cost-effectively by pooling together commodity hardware over the Internet.

However, existing algorithms are not designed to run inference with unreliable devices or high-latency networks. To bridge this gap, we formulate a novel algorithm for fault-tolerant distributed *inference*

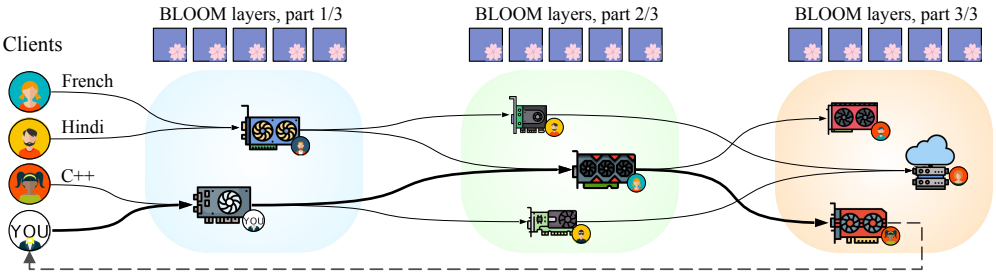

Figure 1: An overview of PETALS system. Servers store pre-trained LLM layers and temporarily hold attention caches for inferencing. Clients hold embedding layers, as well as learned prompts/adapters when inferencing fine-tuned models. Arrows denote temporary chains formed for inference.

of very large models. This algorithm takes care of inference-specific problems: keeping distributed attention caches between inference steps and recovering when some of remote devices fail or leave abruptly. This algorithm allows for several cost-effective ways of using LLMs, such as combining under-utilized GPUs in multiple cloud regions, or forming a collaboration of multiple research groups and connecting their existing infrastructure to run large models together.

The contributions of this work can be summarized as follows:

- We analyze the performance model for training very large language models and identify bottlenecks that are not addressed by existing algorithms. Notably, both local offloading and existing distributed algorithms struggle with sequential generation in different ways.
- We propose a novel distributed algorithm that can inference large (100B+) language models on distributed unreliable devices. To the best of our knowledge, this is the first algorithm that can inference LLMs with 100B+ parameters over the Internet.
- Using this algorithm, we design PETALS – a decentralized system for inferencing and fine-tuning LLMs over the Internet. When inferencing BLOOM-176B, PETALS outperforms offloading by roughly $10\times$ for autoregressive generation. The implementation of our algorithms and PETALS system is available online.[1]

2 BACKGROUND: EFFICIENT TRAINING AND INFERENCE

There is a wide variety of methods that can optimize training and inference for most deep learning workloads. In this section, we focus on two lines of research that are relevant for our analysis: model parallelism and parameter offloading.

2.1 MODEL PARALLELISM

Model parallelism is a family of distributed training algorithms that assigns each device to hold a subset of model parameters, run a subset of computations and communicate output activations. There are two main types of model parallelism: "Traditional" model parallelism and pipeline parallelism. Traditional model parallelism (or tensor parallelism) assigns each device to compute a subset of each model layer (e.g., a subset of neurons), then communicate results between each other and proceed to the next layer (Krizhevsky et al., 2012; Ben-Nun & Hoefler, 2019; Tang et al., 2020). Each device performs a symmetric computation, applied to a different slice of model weights, which makes traditional model parallelism compatible with MPI-based communication. In turn, the main performance overhead of this strategy comes from all-to-all communication (and synchronization) after each layer (Krizhevsky, 2014).

Pipeline parallelism reduces the communication overhead by assigning each device with one or several full layers (Huang et al., 2019; Narayanan et al., 2019; Yang et al., 2019). During the forward pass, each stage applies its subset of layers to the inputs supplied by the previous stage, then sends the outputs of the last layer to the next stage. For the backward pass, this process is reversed, with each pipeline stage passing the gradients to the same device that previously supplied it with input

[1] See `github.com/iclr2023-anonymous/petals`

activations. To better utilize the available devices, the pipeline must process multiple microbatches per step, allowing each stage to run in parallel on a different batch of inputs. Even with optimal execution, some of the pipeline stages will remain idle some of the time (Huang et al., 2019).

Both of these model parallel strategies are still actively used for training LLMs. However, real-world distributed training systems usually combine multiple forms of parallelism depending on hardware and network (Narayanan et al., 2021; Rajbhandari et al., 2020; Jia et al., 2019). In particular, "traditional" model parallelism is typically used within a single multi-GPU server or closely interconnected TPU cores (Narayanan et al., 2021; Shazeer et al., 2018). In turn, pipeline parallelism is used to connect multiple servers(Narayanan et al., 2021). Several recent works demonstrate that pipeline parallelism can be used for cost-efficient *pre-training* of LLMs by pooling together cheap under-utilized GPU devices (Athlur et al., 2022; Kuszmaul, 2022; Wang et al., 2022).

2.2 Offloading

Parameter offloading relegates model parameters from accelerator memory to a slower but cheaper storage: typically RAM or SSD (Pudipeddi et al., 2020; Ren et al., 2021; Rajbhandari et al., 2021). When using the model, parameters are loaded to the accelerator just-in-time for computation, one or few layers at a time. In principle, this method allows running large models with a single low-end accelerator as long as there is enough RAM (or SSD) to store full model parameters.

The main drawback of this strategy is having to load and unload through all model parameters for each forward and backward pass, which can be time-consuming. This extra time can be amortized in workloads where model can do a lot of useful computations for each time a parameter is loaded. In practice, using offloading to run a single token through the OPT-175B on one GPU in the best-case scenario of hardware and bandwidth[2] would require 11 seconds per forward pass, or twice that for training. As we show in Section 4, real-world offloading performance is significantly worse.

Pudipeddi et al. (2020) circumvents this by training with very large batches, and hence, increasing the computation. In turn, Ren et al. (2021); Rajbhandari et al. (2021) reduce the overhead by overlapping communication and computation, that is, doing useful computation for the current layer while waiting for the transfer of the next layer to finish. However, unlike in model-parallel training, Ren et al. (2021) still requires each device to compute the full model after gathering parameters. From the efficiency perspective, the main difference with model parallelism is that, during forward pass, model-parallel algorithms need to transfer layer activations between devices, whereas offloading algorithms transfer model parameters instead.

3 Method

Using pretrained large language models for NLP tasks consists of two main workloads: inference and fine-tuning. The inference workload typically consists of encoding an input text, then generating tokens autoregressively. In turn, fine-tuning requires updating either all of the model's parameters or (more commonly for large models) a small set of trainable weights (e.g., adapters or soft prompts) by backpropagation. These two workloads also cover more advanced use cases, such as:

- Manually engineering prompts for a given task, then deploying the model with these prompts.
- Learning and inference with adapters (Hu et al., 2021; Houlsby et al., 2019; Liu et al., 2022b) or "soft" prompts (Liu et al., 2021b; Lester et al., 2021; Liu et al., 2021a).
- Distillation into a smaller task-specific model for faster inference (Schick & Schütze, 2021).

Counter-intuitively, we found that inference is more challenging than fine-tuning for cost-efficient setups. To that end, we dedicate most of this section to inference-specific problems. As for fine-tuning, we describe a generic way to support arbitrary parameter-efficient fine-tuning in Section 3.3.

3.1 Performance bottlenecks of LLM inference

Unlike training, autoregressive LLM inference cannot be done with a single pass through the model. Instead, the model needs to process one token at a time, pass it through the entire model, then generate

[2] 16-bit parameters, 350GB total, PCIe gen. 4 at 31.5GB/s (16 lanes), infinite compute and memory bandwidth.

the next token and repeat the process. For distributed inferencing with model parallelism, training on a sequence of t tokens needs $O(n)$ communication rounds, but generating the same sequence would require $O(n \cdot t)$ rounds, making it more susceptible to network latency. Similarly with parameter offloading, generating a sequence of t tokens with an n-layer model would require loading every layer t times ($O(n \cdot t)$ too).

The other problem of autoregressive generation is dealing with attention for past tokens Vaswani et al. (2017). During an inference step t, each layer needs to attend to $t - 1$ past keys & values from previously generated tokens. Existing inference algorithms store past entries in accelerator memory. Caching half-precision activations of a 2048-token sequence for large models like GPT-3 (Brown et al., 2020) or OPT-175B (Zhang et al., 2022) (with 96 layers of 12288 units each) will take up 9.6GB GPU memory *for each sequence*. Offloading these cached values faces the same problems as offloading in general.

An alternative solution is to recompute all previous tokens on every inference step, storing only one set of keys & values at a time. Naturally, this approach needs quadratically more computation with sequence length t, for a total of $O(t^3)$ time for transformer-based models since all public LLMs with 100B+ parameters use standard attention, which has a quadratic complexity in terms of sequence length. Surprisingly, this approach is often more efficient than offloaded caching, especially for shorter sequences due to the sheer inefficiency of the latter.

Parameter offloading can still be efficient when generating *large amounts of very short sequences*. Each individual sequence still takes a long time to generate, but the system maintains high throughput by running many samples in parallel. Unfortunately, this is a niche scenario that does not cover most LLM use cases. For instance, it is incompatible with in-context learning or prompt engineering, where the model needs to process long sequences of training examples (Brown et al., 2020). More importantly, it does not support "interactive" applications where LLM needs to quickly respond to a user input. This rules out many LLM applications[3] such as conversation systems, question answering, machine translation or input completion (e.g. "Smart Compose"). To the best of our knowledge, this is a fundamental limitation of offloading that cannot be circumvented with incremental changes.

Hence, we explore a new solution based on pipeline-parallelism. Concurrent work (Aminabadi et al., 2022) uses model parallelism to inference LLMs in GPU clusters. However, their approach does not apply to our more affordable setups: cheap "preemptible" instances or connecting existing resources over the Internet. To operate in these conditions, an inference algorithm needs to deal with node preemption, network errors, and high networks latency.

3.2 Distributed Generation with Fault Tolerance

In this section, we formulate an algorithm for inferencing LLMs in a fleet of unreliable geographically distributed devices connected over the internet. Each device can act as a server or a client: each server holds one or several consecutive transformer blocks, while clients hold token embeddings and run inference jobs. A single device can simultaneously act as client and server, e.g. when several participants connect their existing compute servers to run the same LLM for their own task. For simplicity, we assume that every model layer is hosted on several servers and examine this assumption in the next section. Following this notation, a fault-tolerant algorithm should allow each client to complete their inference job with reproducible results even if any remote servers fail during inference.

As we discuss in Section 3.1, autoregressive generation requires many sequential communication rounds, making it sensitive to network latency. However, if every device stores its past attention cache, every communication round only transfers activations for a single token, i.e. several kilobytes of data[4]. In our algorithm, we use this model to directly minimize the inference time over all possible pipeline configurations. As we show later in Section 4.2, this allows efficient inference over a low-bandwidth Internet connection.

A more challenging problem is how to recover from node and network failures. If a remote server shuts down, any cached attention keys stored on that server will be lost with it. There are two naïve solutions to this problem: restarting inference from scratch or recomputing past embeddings on every step. Restarting might be enough at a small scale. However, running 100B+ models will involve

[3] See e.g. `https://gpt3demo.com`

[4] For GPT-3 and OPT-175B, one 12288-dimensional token embedding takes up exactly 24KB

many unreliable devices, making it unlikely to generate long sequence without at least one failure. In turn recomputing past attention caches requires communicating past tokens on every communication round, resulting in $O(n \cdot t^2)$ total data transferred, where n is the number of pipeline layers and t is the sequence length. In other words, both these solutions struggle to generate long sequences.

We address this problem by maintaining two types of cache: *server-side cache* holds past attention keys and values for their layers, while *client-side cache* holds past inputs sent to a given pipeline stage. If a server disconnects, a client can find another server with that pipeline stage and use client-side cache to restore the server state.

The resulting procedure is described in Algorithm 1. For every pipeline stage, the client maintains a heap (priority queue) of servers that hold this stage (and may hold additional stages). The servers in queue are ordered by the network latency, measured by ping. Crucially, the queues are maintained through the lifetime of a client. To begin generation, the client runs a beam-search-like procedure to find a sequence of servers that results in the least total inference time under our performance model. When running inference steps, a client keeps track of intermediate activations sent between pipeline stages. If one of remote servers fails or leaves, the client retrieves the next best server (or multiple servers) from the heap and directs it to restore the attention state from client's cached activations.

Algorithm 1 Generating sequence, client-side code

```
Input: prefix_tokens, embeddings, known_servers
generated_sequence = list()
cache = dictionary()
streams = dictionary()
chain = find_best_chain(known_servers)
for server ∈ chain do
    streams[server] = rpc_inference(server)
    cache[server] = list()

inputs = embeddings(prefix_tokens)
while should_continue(generated_sequence) do
    tail_servers = copy(chain)
    while not empty(tail_servers) do
        server = tail_servers.pop_left()
        try:
            ▷ Attempt normal inference
            outputs = streams[server].send(inputs)
            cache[server].append(inputs)
            inputs = outputs
        catch FailedRPC:
            ▷ Replace the failed server
            streams.pop(server).close()
            past_inputs = cache.pop(server)
            new_servers = replace_failed_server(
                server, past_inputs, cache,
                streams, known_servers)
            chain.replace(server, new_servers)
            tail_servers.push_left(new_servers)

    logits = compute_logits(outputs, embeddings)
    next_token = choose_next(logits)  ▷ e.g. greedy
    generated_sequence.append(next_token)
    inputs = embeddings(next_token)

for server ∈ chain do
    streams[server].close()
return generated_sequence
```

Algorithm 2 rpc_inference(server)

```
Input: local_layers, stream
cache = dictionary()
for layer ∈ local_layers do
    cache[layer] = make_empty()
while not stream.closed() do
    inputs = stream.receive()
    for layer ∈ local_layers do
        past_kv = cache[layer]
        inputs, new_kv = forward(
            layer, inputs, past_kv)
        cache[layer].append(new_kv)
    stream.send(inputs)
```

Algorithm 3 replace_failed_server(...)

```
Input: server, inputs, cache, streams,
    known_servers
known_servers.ban(server)
missing_layers = get_layers(server)
candidates = select_by_layer(
    known_servers, missing_layers)
chain = find_best_chain(candidates)
replacements = list()
while not empty(chain) do
  s = chain.pop_left()
  try:
    streams[s] = rpc_inference(s)
    outputs = streams[s].send(inputs)
    replacements.append(s)
    cache[s] = inputs
    missing_layers.pop(get_layers(s))
    inputs = outputs
  catch FailedRPC:
    known_servers.ban(s)
    candidates = select_by_layer(
      candidates, missing_layers)
    chain = find_best_chain(candidates)
return chain
```

When a remote server fails, the algorithm needs to send $O(t)$ data (in one round) for each failed server and compute only the stages held by the failed server. This can be seen as an interpolation between naive and cached inference, depending on the server failure rate. If none of the servers fail, we recover $O(n \cdot t)$ communication, similar to Aminabadi et al. (2022). In turn, if all servers fail after one step, the algorithm effectively performs non-caching generation, which is the best option in that scenario.

In practice, it is possible to modify the algorithm to further reduce the number of network hops. In the basic formulation, all communication between pipeline stages is routed through the client, i.e. the client receives the outputs of every pipeline stage, caches it and sends the to the subsequent stage. In practice, it is more efficient to let pipeline stages communicate directly: once the server obtains output activations, it sends them to both client and the subsequent stage. This reduces the total step time since both messages are a few kilobytes in size an can be sent in parallel. To verify that both client and the next pipeline stage received the same set of activations, they can verify the checksums (i.e. hash values) of the received activations asynchronously, without blocking computation.

Algorithm 1 can support greedy inference or any sampling variants (including Holtzman et al. (2020)). However, it requires one more step to support search-based algorithms: cache reordering. During inference step, a client can specify which attention state (from a beam) to use for every new token, possibly using some states more than once.

3.3 System design

So far, we have focused on whether it is possible to run LLM inference on geo-distributed unreliable devices. However, this does not mean that it is practical for a researcher to do so. For example, before using Algorithm 1, each server must be assigned to a pipeline stage — and then reassigned each time another server joins and leaves the network. Here, we describe the design and implementation details that make the system practical. Our description centers around Petals - a decentralized system that allows geographically distributed devices to run 100B+ parameter language models.

We design Petals to run persistently across multiple projects, using a fleet of servers that can join or leave the system at any time. To operate in these conditions, a system should be able to automatically reassign layers as new servers join or leave. Furthermore, the system should support multiple clients using the model simultaneously for different tasks, e.g. when one client is inferencing the base model, another could be fine-tuning it to a different task or inferencing a previously fine-tuned task. Last but not least, it should be easy to use for popular workloads, i.e. run clients and servers.

Server load balancing. As we state earlier, the algorithm relies on having multiple servers per pipeline stage. To that end, each server runs a load balancing procedure to select which layers it needs to serve. Formally, servers maximize the total model throughput by choosing the blocks with the worst throughput and eliminating potential bottlenecks. Each server periodically announces its active blocks to a distributed hash table (Maymounkov & 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.

Parameter efficient fine-tuning. 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 the model, gradients, and optimizer states.

Fortunately, *parameter-efficient fine-tuning* methods have been developed that keep most of the pretrained model intact. Some of them choose a subset of existing parameters to update (Sung et al.,

2021; Guo et al., 2021) while others augment the model with additional trainable weights (Hu et al., 2021; Houlsby et al., 2019; Liu et al., 2021b; Lester et al., 2021; Liu et al., 2021a; 2022b). 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 & Nikoulina, 2022) and even outperform it in low-data regimes (Liu et al., 2022a). Another appealing property of these approaches for our use-case is that they allow rapidly switching a pretrained LLM between different adapters.

By focusing on parameter-efficient fine-tuning, we are able to simplify the design of PETALS by *making clients hold all learned parameters* (see Figure 1). Servers can run backpropagation through their layers and return gradients with respect to activations, but they *do not update the server-side parameters*. Even when client communicates learned values (e.g. soft prompts) to a server, the server treats these values same as input activations. Thus, a server can simultaneously run different fine-tuning tasks without them interfering with one another. This design choice also allows PETALS users to define custom adapters in simple PyTorch without the need for network engineering expertise. We discuss the design of the client-side API in more detail for inference and fine-tuning in Appendix A.

Memory efficiency. Since our main intended use-case is running on inexpensive low-end devices, we need to work around their capabilities. 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.

More specifically, 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. We verify that compressing weights does not affect model quality in Appendix B.

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. Dynamic blockwise quantization halves the bandwidth requirements without any noticeable effect on generation quality. When fine-tuning, we also take advantage of gradient checkpointing (Griewank & Walther, 2000; Chen et al., 2016) and half precision to reduce VRAM usage — both of which are standard practice for large language models (Narayanan et al., 2021; Brown et al., 2020; Athlur et al., 2022). In experiments, we also apply these optimizations to baseline systems for a fair comparison.

4 EXPERIMENTS

4.1 INFERENCE WITH UNRELIABLE SERVERS

First, we conduct small-scale preliminary experiments to test the fault-tolerant generation algorithm described in Section 3.2. For these experiments, we use a smaller BLOOM model with 7.1 billion parameters[5]. This model consists of 30 transformer blocks with hidden size 4096. We compare our algorithm with baselines when generating a single sequence of length 512 autoregressively. For simplicity, we run all computations and communications in single precision and disregard word embeddings / logits for this set of experiments. We measure the time to run a certain number of tokens through all transformer layers. We simulate network failures by resetting pipeline stages at a certain rate.

We compare three different inference strategies:

1. Standard inference with attention cache that restarts if cache is lost.
2. Stateless inference that reruns past tokens on every step with no need to restart.
3. Fault-tolerant inference, as described in Algorithm 1.

[5]See `https://huggingface.co/bigscience/bloom-7b1`

Table 1: Sequential inference speed (steps per second) of BLOOM-7B1 with varying failure rates. The missing value for caching with restarts indicates that the algorithm did not finish in tiple hours

Inference Algorithm	**128 tokens, failure rate:** 0	1e-4	1e-3	1e-2	**1024 tokens, failure rate:** 0	1e-4	1e-3	1e-2
Caching with restarts	17.1	16.7	12	0.18	15.5	11.8	0.48	–
No caching (recompute)	3.44	3.44	3.44	3.44	0.89	0.89	0.89	0.89
Algorithm 1	11.4	11.4	10.6	3.38	10.7	10.7	7.76	2.17

All runs use four pipeline stages with (8, 7, 8, 7) model layers per pipeline stage. Each pipeline stage is served by a single GeForce 1080Ti GPU; the four GPUs are running in a single system with dual Xeon Gold 6148 CPU, 12 DDR4 LRDIMM sticks with 64GB each. The system has 16 dedicated PCIe Gen.3 lanes per GPU in dual root configuration, without using PCIe switches. Each stage runs in an isolated docker containers with virtual network interfaces, but there is no limit to communication bandwidth for this experiment. We repeat all experiments 50 times and report the average time. The adjusted standard deviation never exceeds 0.2%. For baselines, we use use the pipeline parallelism implementation from Megatron-DeepSpeed[6] for the stateless inference baseline.

We report all performance measurements in Table 1. Our algorithm outperforms both baselines for setups with high failure rate. Caching with restarts is most efficient for inferencing with no failures. We also note that our algorithm is somewhat slower than this baseline when there are no failures. We attribute this to a more technically advanced implementation of that baseline. Finally, non-caching inference can be competitive for short sequences (128 tokens), but slows down considerably on 1024 tokens, which agrees with out intuition from 3.1.

4.2 Experiments for BLOOM-176B

Next, we evaluate PETALS on a more practical task of running BLOOM-176B[7] - a Transformer language model containing 70 layers with a hidden size of 14336. We evaluate three server configurations running in a network with controlled bandwidth. Our first setup consists of 3 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). In total, there are 9 virtual servers running 7 blocks each, one server with 3 blocks and two more servers with 2 blocks.

Finally, we evaluate inference with parameter offloading, where each user runs independently on a single GPU, swapping parameters from CPU memory. We report two offloading values: the real-world throughput with DeepSpeed with the default recommended parameters, but enable `pin_memory`, since we found that this settings runs $1.2{-}2\times$ faster in all setups. For theoretical best throughput, we calculate the maximum inference and forward training throughput to receive an upper bound on offloading performance. For more details of this estimate, see Appendix C.

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

In Table 2, we report the performance of sequential inference and parallel forward passes. For inference, performance does not depend much on bandwidth or sequence length but degrades in high-latency settings, especially for 12 virtual servers. In turn, training-time forward passes for large batches are affected by both bandwidth and latency. These results are shown in Table 2. 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.

[6]See `https://github.com/bigscience-workshop/Megatron-DeepSpeed`

[7]See `https://huggingface.co/bigscience/bloom`

[8]We set network conditions with `https://github.com/magnific0/wondershaper`, based on `tc qdisc`

Table 2: Throughput and latency for BLOOM-176B generation and forward pass

Network		Inference (steps/s)		Forward (tokens/s)	
		Sequence length		Batch size	
Bandwidth	**Latency**	128	2048	1	64
Offloading on 1x A100 (best-case theoretical estimates and actual)					
128 Gbit/s	–	0.09	0.09	2.4	152.8
256 Gbit/s	–	0.18	0.18	2.7	170.3
Actual	–	0.0485	0.0495	2.5	152.4
PETALS on 3 physical servers, with one A100 each					
1 Gbit/s	< 5 ms	1.22	1.11	70.0	253.6
100 Mbit/s	< 5 ms	1.19	1.08	56.4	182.0
100 Mbit/s	100 ms	0.89	0.8	19.7	112.2
PETALS on 12 virtual servers, simulated on 3x A100					
1 Gbit/s	< 5 ms	0.97	0.86	37.9	180.0
100 Mbit/s	< 5 ms	0.97	0.86	25.6	66.6
100 Mbit/s	100 ms	0.44	0.41	5.8	44.3
Same, but with 8 clients running simultaneously					
1 Gbit/s	< 5 ms	0.75	0.72	–	–
100 Mbit/s	< 5 ms	0.74	0.7	–	–
100 Mbit/s	100 ms	0.36	0.35	–	–
PETALS on 14 real servers in Europe and North America					
Real world		0.68	0.61	32.6	179.4

4.3 RUNNING BLOOM-176B OVER THE INTERNET

Finally, we benchmark BLOOM in a real-world distributed setting with 14 smaller servers holding 2×RTX 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 behind firewalls[9].

We evaluate this configuration for the same tasks as in 4.2 and report results at the bottom of Table 2. This setup is marginally slower than A100 benchmarks, which is expected due to slower hardware. However, even when communicating between different continents, PETALS maintains its efficiency, still outperforming offloading by a large margin.

5 CONCLUSION

In this paper, we introduced a novel fault-tolerant algorithm for inferencing large language models. On top of it, we introduced PETALS – a decentralized system for running LLMs on distributed unreliable devices connected over the Internet, which significantly outperforms other approaches to running inference on consumer-grade hardware. We demonstrated that the proposed system can scale to the largest publicly available langauge model with hundreds of billions of trainable parameters.

While our work is focused on technical aspects, running large language models over the Internet raises a broad range of related questions. One important consideration is privacy: making sure that using PETALS does not leak private data to outside peers. Last but not least, we need 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, incentive structures, and further "evolution" of distributed models in Appendix D.

[9]We use the Circuit Relay protocol from libp2p to traverse NATs and firewalls, see `https://docs.libp2p.io/concepts/circuit-relay/`

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. Consequently, 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 that can be served with PETALS are already in open access and thus can be exposed via APIs or other means: we do not release any artifacts beside the code.

One important limitation of our work in its current state 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. We discuss this limitation in more detail in Appendix D and acknowledge that the development of methods for privacy-preserving decentralized inference without performance penalties is an open research problem.

REFERENCES

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

Reza Yazdani Aminabadi, Samyam Rajbhandari, Minjia Zhang, Ammar Ahmad Awan, Cheng Li, Du Li, Elton Zheng, Jeff Rasley, Shaden Smith, Olatunji Ruwase, et al. Deepspeed inference: Enabling efficient inference of transformer models at unprecedented scale. *arXiv preprint arXiv:2207.00032*, 2022.

Sanjith Athlur, Nitika Saran, Muthian Sivathanu, Ramachandran Ramjee, and Nipun Kwatra. Varuna: scalable, low-cost training of massive deep learning models. In *Proceedings of the Seventeenth European Conference on Computer Systems*, pp. 472–487, 2022.

Tal Ben-Nun and Torsten Hoefler. Demystifying parallel and distributed deep learning: An in-depth concurrency analysis. *ACM Comput. Surv.*, 52(4), aug 2019. ISSN 0360-0300. doi: 10.1145/3320060. URL https://doi.org/10.1145/3320060.

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. Improving language models by retrieving from trillions of tokens, 2021. URL https://arxiv.org/abs/2112.04426.

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

Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. Training deep nets with sublinear memory cost. *arXiv preprint arXiv:1604.06174*, 2016.

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

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

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. Glam: Efficient scaling of language models with mixture-of-experts. *CoRR*, abs/2112.06905, 2021. URL https://arxiv.org/abs/2112.06905.

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

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

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

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. A framework for few-shot language model evaluation, September 2021. URL https://doi.org/10.5281/zenodo.5371628.

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. Gemv2: Multilingual nlg benchmarking in a single line of code, 2022.

Andreas Griewank and Andrea Walther. Algorithm 799: revolve: an implementation of checkpointing for the reverse or adjoint mode of computational differentiation. *ACM Transactions on Mathematical Software (TOMS)*, 26(1):19–45, 2000.

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

Ari Holtzman, Jan Buys, Li Du, Maxwell Forbes, and Yejin Choi. The curious case of neural text degeneration. In *International Conference on Learning Representations*, 2020. URL https://openreview.net/forum?id=rygGQyrFvH.

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

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

Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, et al. Gpipe: Efficient training of giant neural networks using pipeline parallelism. In *Advances in Neural Information Processing Systems*, pp. 103–112, 2019.

Zhihao Jia, Matei Zaharia, and Alex Aiken. Beyond data and model parallelism for deep neural networks. In A. Talwalkar, V. Smith, and M. Zaharia (eds.), *Proceedings of Machine Learning and Systems*, volume 1, pp. 1–13, 2019. URL https://proceedings.mlsys.org/paper/2019/file/c74d97b01eae257e44aa9d5bade97baf-Paper.pdf.

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

Michael Khrushchev, Ruslan Vasilev, Nikolay Zinov, Alexey Petrov, and Yandex. Yalm 100b, 2022. "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. Dynabench: Rethinking benchmarking in nlp, 2021. URL https://arxiv.org/abs/2104.14337.

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. What changes can large-scale language models bring? intensive study on hyperclova: Billions-scale korean generative pretrained transformers. *CoRR*, abs/2109.04650, 2021. URL https://arxiv.org/abs/2109.04650.

Alex Krizhevsky. One weird trick for parallelizing convolutional neural networks. *CoRR*, abs/1404.5997, 2014. URL http://arxiv.org/abs/1404.5997.

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger (eds.), *Advances in Neural Information Processing Systems 25*, pp. 1097–1105. Curran Associates, Inc., 2012. URL http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf.

John Kuszmaul. Bamboo trimming revisited: Simple algorithms can do well too. *arXiv preprint arXiv:2201.07350*, 2022.

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

Brian Lester, Rami Al-Rfou, and Noah Constant. The power of scale for parameter-efficient prompt tuning. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pp. 3045–3059, Online and Punta Cana, Dominican Republic, November 2021. Association for Computational Linguistics. doi: 10.18653/v1/2021.emnlp-main.243. URL https://aclanthology.org/2021.emnlp-main.243.

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. Retrieval-augmented generation for knowledge-intensive nlp tasks. In H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin (eds.), *Advances in Neural Information Processing Systems*, volume 33, pp. 9459–9474. Curran Associates, Inc., 2020. URL https://proceedings.neurips.cc/paper/2020/file/6b493230205f780e1bc26945df7481e5-Paper.pdf.

Xiang Lisa Li and Percy Liang. 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)*, pp. 4582–4597, Online, August 2021. Association for Computational Linguistics. doi: 10.18653/v1/2021.acl-long.353. URL https://aclanthology.org/2021.acl-long.353.

Haokun Liu, Derek Tam, Mohammed Muqeeth, Jay Mohta, Tenghao Huang, Mohit Bansal, and Colin Raffel. Few-shot parameter-efficient fine-tuning is better and cheaper than in-context learning, 2022a. URL https://arxiv.org/abs/2205.05638.

Haokun Liu, Derek Tam, Mohammed Muqeeth, Jay Mohta, Tenghao Huang, Mohit Bansal, and Colin Raffel. Few-shot parameter-efficient fine-tuning is better and cheaper than in-context learning, 2022b. URL https://arxiv.org/abs/2205.05638.

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

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

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

Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R. Devanur, Gregory R. Ganger, Phillip B. Gibbons, and Matei Zaharia. Pipedream: Generalized pipeline parallelism for dnn training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, pp. 1–15, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450368735. doi: 10.1145/3341301.3359646. URL https://doi.org/10.1145/3341301.3359646.

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

NVIDIA. Nvidia ampere ga102 gpu architecture, 2020. URL https://images.nvidia.com/aem-dam/en-zz/Solutions/geforce/ampere/pdf/NVIDIA-ampere-GA102-GPU-Architecture-Whitepaper-V1.pdf.

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

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

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

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

Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. Improving language understanding by generative pre-training. 2018. URL https://cdn.openai.com/research-covers/language-unsupervised/language_understanding_paper.pdf.

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

Jack W. Rae, Sebastian Borgeaud, Trevor Cai, Katie Millican, Jordan Hoffmann, H. Francis Song, John Aslanides, Sarah Henderson, Roman Ring, Susannah Young, Eliza Rutherford, Tom Hennigan, Jacob Menick, Albin Cassirer, Richard Powell, George van den Driessche, Lisa Anne Hendricks, Maribeth Rauh, Po-Sen Huang, Amelia Glaese, Johannes Welbl, Sumanth Dathathri, Saffron Huang, Jonathan Uesato, John Mellor, Irina Higgins, Antonia Creswell, Nat McAleese, Amy Wu, Erich Elsen, Siddhant M. Jayakumar, Elena Buchatskaya, David Budden, Esme Sutherland, Karen Simonyan, Michela Paganini, Laurent Sifre, Lena Martens, Xiang Lorraine Li, Adhiguna Kuncoro, Aida Nematzadeh, Elena Gribovskaya, Domenic Donato, Angeliki Lazaridou, Arthur Mensch, Jean-Baptiste Lespiau, Maria Tsimpoukelli, Nikolai Grigorev, Doug Fritz, Thibault Sottiaux, Mantas Pajarskas, Toby Pohlen, Zhitao Gong, Daniel Toyama, Cyprien de Masson d'Autume, Yujia Li, Tayfun Terzi, Vladimir Mikulik, Igor Babuschkin, Aidan Clark, Diego de Las Casas, Aurelia Guy, Chris Jones, James Bradbury, Matthew Johnson, Blake A. Hechtman, Laura Weidinger, Iason Gabriel, William S. Isaac, Edward Lockhart, Simon Osindero, Laura Rimell, Chris Dyer, Oriol Vinyals, Kareem Ayoub, Jeff Stanway, Lorrayne Bennett, Demis Hassabis, Koray Kavukcuoglu, and Geoffrey Irving. Scaling language models: Methods, analysis & insights from training gopher. *CoRR*, abs/2112.11446, 2021. URL https://arxiv.org/abs/2112.11446.

Colin Raffel. A call to build models like we build open-source software, 2021. URL https://colinraffel.com/blog/a-call-to-build-models-like-we-build-open-source-software.html.

Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. Zero: Memory optimization towards training a trillion parameter models. In *SC*, 2020.

Samyam Rajbhandari, Olatunji Ruwase, Jeff Rasley, Shaden Smith, and Yuxiong He. 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*, SC '21, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450384421. doi: 10.1145/3458817.3476205. URL https://doi.org/10.1145/3458817.3476205.

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

Timo Schick and Hinrich Schütze. Generating datasets with pretrained language models. pp. 6943–6951, November 2021. doi: 10.18653/v1/2021.emnlp-main.555. URL https://aclanthology.org/2021.emnlp-main.555.

Noam Shazeer, Youlong Cheng, Niki Parmar, Dustin Tran, Ashish Vaswani, Penporn Koanantakool, Peter Hawkins, HyoukJoong Lee, Mingsheng Hong, Cliff Young, Ryan Sepassi, and Blake A. Hechtman. Mesh-tensorflow: Deep learning for supercomputers. *CoRR*, abs/1811.02084, 2018. URL http://arxiv.org/abs/1811.02084.

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

Zhenheng Tang, Shaohuai Shi, Xiaowen Chu, Wei Wang, and Bo Li. Communication-efficient distributed deep learning: A comprehensive survey, 2020.

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

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. Attention is all you need. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (eds.), *Advances in Neural Information Processing Systems 30*, pp. 5998–6008. Curran Associates, Inc., 2017. URL http://papers.nips.cc/paper/7181-attention-is-all-you-need.pdf.

Jue Wang, Binhang Yuan, Luka Rimanic, Yongjun He, Tri Dao, Beidi Chen, Christopher Re, and Ce Zhang. Fine-tuning language models over slow networks using activation compression with guarantees, 2022. URL https://arxiv.org/abs/2206.01299.

Peter West, Chandra Bhagavatula, Jack Hessel, Jena D Hwang, Liwei Jiang, Ronan Le Bras, Ximing Lu, Sean Welleck, and Yejin Choi. Symbolic knowledge distillation: from general language models to commonsense models. *arXiv preprint arXiv:2110.07178*, 2021.

Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi 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 M. Rush. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pp. 38–45, Online, October 2020. Association for Computational Linguistics. URL https://www.aclweb.org/anthology/2020.emnlp-demos.6.

Bowen Yang, Jian Zhang, Jonathan Li, Christopher Ré, Christopher R. Aberger, and Christopher De Sa. Pipemare: Asynchronous pipeline parallel dnn training. *ArXiv*, abs/1910.05124, 2019.

Zheng-Xin Yong and Vassilina Nikoulina. Adapting bigscience multilingual model to unseen languages, 2022. URL https://arxiv.org/abs/2204.04873.

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. GLM-130B: An open bilingual pre-trained model, 2022. URL http://keg.cs.tsinghua.edu.cn/glm-130b/posts/glm-130b/.

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. Pangu-α: Large-scale autoregressive pretrained chinese language models with auto-parallel computation. *CoRR*, abs/2104.12369, 2021. URL https://arxiv.org/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. Opt: Open pre-trained transformer language models, 2022. URL https://arxiv.org/abs/2205.01068.

APPENDIX

A 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.

A.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.

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.

A.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

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

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

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

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 choose a subset of existing parameters (Sung et al., 2021; Guo et al., 2021), others augment the model with extra trainable weights (Hu et al., 2021; Houlsby et al., 2019; Liu et al., 2021b; Lester et al., 2021; Liu et al., 2021a; 2022b).

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 & Nikoulina, 2022) and even outperform it in low-data regimes (Liu et al., 2022a). 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 A.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 & 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).

```
# 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()
```

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

Table 3: 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 4: 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

A.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.

B Quality and efficiency of BLOOM with 8-bit quantization

As shown in Table 3, this method has little effect on LLM quality for major benchmarks. In terms of inference time, Table 4 demonstrates that quantization has about 5% of overhead with batch size 1 (20 tokens), but becomes negligible for larger batches.

C Estimating theoretical best throughput with RAM offloading

In this estimate, we use the best possible hardware setup for offloading: CPU RAM offloading via PCIe 4.0 with 16 PCIe lanes per GPU. 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 and 128 Gbit/s for PCIe 3.0. As such, offloading 176B parameters takes 5.5 seconds for PCIe 4.0 and 11 seconds for PCIe 3.0. We assume an offloading latency of zero for the upper bound estimation. In reality, many GPU servers use PCIe switches which limits the bandwidth. For instance, if two GPUs are behind a PCIe switch, transfering data to both at the same time will halve the bandwidth.

D Discussion and future work

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).

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. This system may be implemented using a centralized "accounting" server or in a decentralized way.

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 A.2, distributed parameter-efficient fine-tuning makes it easy for users to apply the base model to new tasks. In Section A.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. Similarly, fine-tuned model adapters could be annotated with tags denoting the model version they are applicable for. Such fine-grained versioning of models is currently uncommon but would be straightforward to add to PETALS.