DiPaCo: Distributed Path Composition

Arthur Douillard[*,1], **Qixuan Feng**[*,1], **Andrei A. Rusu**[*,1], **Adhiguna Kuncoro**[1], **Yani Donchev**[1], **Rachita Chhaparia**[1], **Ionel Gog**[1], **Marc'Aurelio Ranzato**[◇,1], **Jiajun Shen**[◇,1] **and Arthur Szlam**[◇,1]
[*]Equal core contributions, [◇]Equal leading contributions, [1]Google DeepMind

Progress in machine learning (ML) has been fueled by scaling neural network models. This scaling has been enabled by ever more heroic feats of engineering, necessary for accommodating ML approaches that require high bandwidth communication between devices working in parallel. In this work, we propose a co-designed modular architecture and training approach for ML models, dubbed DIstributed PAth COmposition (DiPaCo). During training, DiPaCo distributes computation by paths through a set of shared modules. Together with a Local-SGD inspired optimization (DiLoCo) that keeps modules in sync with drastically reduced communication, Our approach facilitates training across poorly connected and heterogeneous workers, with a design that ensures robustness to worker failures and preemptions. At inference time, only a single path needs to be executed for each input, without the need for any model compression. We consider this approach as a first prototype towards a new paradigm of large-scale learning, one that is less synchronous and more modular.
Our experiments on the widely used C4 benchmark show that, for the same amount of training steps but less wall-clock time, DiPaCo exceeds the performance of a 1 billion-parameter dense transformer language model by choosing one of 256 possible paths, each with a size of 150 million parameters.

Keywords: modularity, large-scale, mixture of experts, language modeling, distributed learning

1. Introduction

Progress in machine learning and AI has been driven by spending more FLOPs on larger neural network models, trained on bigger data sets. This scaling has been accomplished via data and model parallelism (Dean et al., 2012) and pipelining (Narayanan et al., 2020) to distribute computation, enabling the concurrent use of a large number of devices (Anil et al., 2023; OpenAI et al., 2023; Touvron et al., 2023). Although model architectures (Lepikhin et al., 2021; OpenAI et al., 2023) have also been used to allow computational parallelism, and optimization procedures to prefer larger batches (Goyal et al., 2017) (again allowing more data parallelism), the current training paradigm has not fundamentally changed model architecture or optimization procedure to facilitate distributed training. State of the art models are still essentially monoliths, and their optimization requires exchanges of parameters, gradients, and activations at every step of the learning process.

This approach incurs engineering and infrastructure challenges associated with provisioning and managing the large number of tightly interconnected devices required for the lengthy training process. The training process itself is often restarted for each new model release, essentially discarding much of the computation for training the last model. Moreover, training monoliths incurs human-organizational challenges, as it is difficult to localize the effects to the final model of changes to any step in the process (beyond data preparation). In particular, it is difficult to leverage the potential of the greater ML community, rather than a single organization (Raffel, 2023b). Because of these challenges, it may become more difficult to continue to scale with the current approach.

As in (Barham et al., 2022; Borzunov et al., 2022; Raffel, 2023b; Ryabinin and Gusev, 2020) and depicted in Figure 1, we envision an alternative approach where models are scalable both in terms of ability to ingest and train on large amounts of data and in terms of enabling collaboration with many contributors, and can be continuously updated and expanded, by virtue

Corresponding author(s): douillard@google.com

of modular design. Modular ML may have many other benefits, see Pfeiffer et al. (2023) for further discussion and references.

In this work, we take a step towards this scalable modular ML paradigm, proposing an architecture and training algorithm, DIstributed PAths COmposition (DiPaCo). DiPaCo's architecture and optimization have been co-designed to reduce communication and enable better scaling. The high level idea is to *distribute computation by path*; here, a "path" means a sequence of modules that define an input-output function. Paths are small relative to the entire model, thus requiring only a handful of tightly connected devices to train or evaluate. During both training and deployment, a query is routed to a replica of a path rather than a replica of the whole model; in other words, the DiPaCo architecture is sparsely activated. We give a detailed exposition of DiPaCo in section 2, and the infrastructure we used to implement it in section 3.

In section 4 we demonstrate the feasibility of DiPaCo by training a language model on the C4 dataset (Raffel et al., 2020) with paths of size 150 million parameters, matching the performance in terms of validation perplexity of a 1.3 billion model, but with 45% less wall clock training time. While the dense 1.3B system required the use of all co-located devices, DiPaCo uses 256 islands of compute, each of which is one-eighth the number of devices used to train the baseline. Neither during train or evaluation time is it necessary to co-locate the full DiPaCo model.

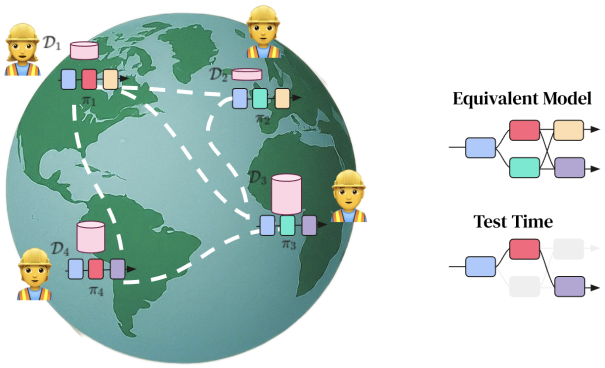

Figure 1 | **Long-term Goal**: Ultimately, we envision a modular network where different components, *paths* π_i, are optimized for different tasks, $\mathcal{D}_j$, each designed by different researchers. The paths, trained on any available hardware type, communicate infrequently across the world, exchanging useful information and enabling new forms of composition.

2. Approach

In this section we give a detailed description of DiPaCo. We start by describing the assumptions and setting in which we will work.

2.1. Setting

Our goal in this work is to demonstrate scalable ML models in the setting of many smaller islands of compute, as opposed to one tightly connected compute landmass. Accordingly, we assume:
1. Training compute (FLOPs) is relatively cheap
2. Communication is relatively expensive

These assumptions are *not* realistic in the current ML training paradigm. As discussed above, devices are usually co-located and run by a single organization, FLOPs are expensive, constrained by the purchase and installation of many accelerators, but communication between devices is relatively cheap because of the co-location.

Nevertheless, in our view, these assumptions may be realistic in the near future, if our compute requirements (or model sizes) grow beyond what can be reasonably co-located, and so communication costs become relatively more expensive. Conversely, progress in distributed training of ML models may allow simpler infrastructure buildouts, leading eventually to more available compute. As it is, infrastructure is designed around the standard approach to training large monoliths; and ML models are architected to take advantage of the current infrastructure and training approaches. This feedback loop may be leading the community to a spurious local minimum where compute is more constrained than it needs to be.

In addition (and closely related) to the two assumptions above, we also assume that both during train and evaluation, we cannot instantiate

models as large as we would like to have on any single compute island.

We will work in the context of language modeling (LM). Based on the above assumptions, we will evaluate models by evaluation perplexity (PPL) against wall-clock time, keeping evaluation complexity in mind . We choose this metric because it encourages exploring how to distribute model training, and more generally, points to in our opinion plausible new paradigms for ML.

2.2. Overview of System

The core idea of DiPaCo is to train a sparsely-activated modular system where data and computation are distributed by the choice of path through the modules. There are two key ideas to making this work:

Coarse Routing: Sparsely routed Mixture of Experts (MoE) have shown great results in language modeling (Lepikhin et al., 2021). The standard approach in a transformer MoE LM is to make a routing decision at each token, based on the feature at each routed layer. In contrast, in this work, during training we will route once per document and offline, as in (Gross et al., 2017; Gururangan et al., 2023).

Routing once per document allows batching computation across all tokens of a sequence, without the need to swap modules in and out as a sequence is processed. This in turn allows parameters to be distributed across distant workers. In addition, instead of learning the router along with the model, we will compute routing decisions *offline*. This enables pre-sharding data by path, before the start of training. This is critical to distribute training across paths, as each worker can now train a path by processing its own shard of data, using its own set of parameters. In subsection 2.4, we will describe in detail how we approach coarse routing.

DiLoCo to keep modules in sync Paths cannot be trained completely independently, because some modules might be shared across multiple paths. To support module sharing across paths, we use DiLoCo (Douillard et al., 2023) for low communication data parallelism, see subsection 2.5. If there are P paths (assigned to P workers) that share module i, each corresponding worker performs SGD on its own shard of data, and every *few hundred* steps, workers average the difference of the parameters of module i before and after the local SGD phase. These averages are then used to update a global parameter vector which is then re-distributed across the workers to sync them, a procedure described in subsection 2.6.

With these two choices, neither at training nor at test time does the entire network (collection of paths) need to be materialized together, as shown in Figure 1.

2.3. Notation

In this section we introduce the basic notation used throughout this work. We assume that we have a base model architecture with parameters θ. We take a partition of the parameter indices into L subsets $B_l, l \in [1, \ldots, L]$. For each l, we choose a number K_l that represents the number of possible choices for the parameters in B_l. We will call a set of parameters associated to a B_l a "module" or, as in (Jacobs et al., 1991; Jordan and Jacobs, 1994), an "expert". In this terminology, K_l is the number of distinct modules associated to B_l.

For a simple example, consider a four-layer fully connected network A. We might take the first two layers to be B_1 and the third and fourth layers to be B_2, so $L = 2$. If we choose $K_i = 3$ for $i = 1, 2$, then we have a 3 modules in B_1, and 3 modules in B_2, for a total of 6 modules. Any choice of module from B_1 and module form B_2 defines a neural network; as in Dean (2021), we call each of these 9 possible networks a "path", see Figure 2.

In this example, and most of the cases we will consider in the rest of this paper, the "blocks" B_i determine contiguous sub-networks of A that can act as an input-output mapping, but this is not necessary. We could just as easily let B_1 be layers 1 and 3 and B_2 be layers 2 and 4; or B_1 be all the biases in the network, and B_2 be all the linear

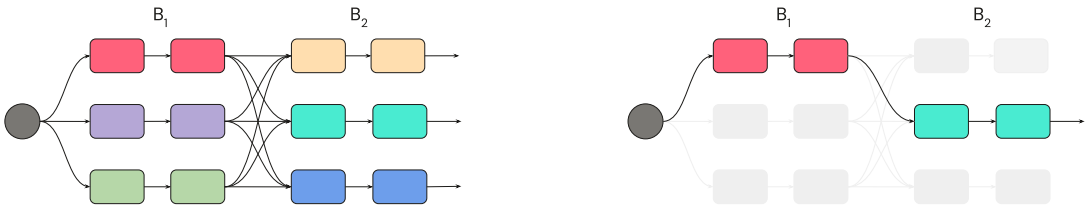

Figure 2 | An illustration of the first example from Section 2.3. A 4 layer neural network, with block B_1 consisting of the first 2 layers and B_2 consisting of the next 2 layers. Each block has 3 choices of module (each with its own parameters), represented by different colors. On the left, we show all of the 9 possible paths. On the right, we show a single path.

parameters. Nevertheless, we will call i the "level" of the module.

In order to transform an input x to an output y, we need to figure out which parameters will be used to operate on x. The function that takes x to a choice of parameters will be called the "router", and is denoted by r; thus $r(x)$ has the form $[j_1, ..., j_L]$, where each j_l indexes one of the K_l choices of parameters for B_l, and r maps an input to a path $\pi = \pi_{j_1,...,j_L}$. We will also often collapse the sequence $j_1, ..., j_L$ into a single number j, where $j \in \{1, \ldots, P = \Pi_{i=1}^{L} K_i\}$. If the training dataset is denoted by X, the subset of data that is routed to path j will be called the j-th "shard" $\mathcal{D}_j$.

model decompose into J paths, data decompose into J shards, and each shard of data used to train each path, shared elements in various paths get assembled

When the router maps an x to a distribution over $\{1, \ldots, P\}$ rather to a deterministic choice, we say that the router is "soft". In this work, we will consider "hard" routers instead, where the choice is single valued and deterministic.

Finally, it is useful to refer to module parameters. Assume that the i-th path goes through module e at level l, with $e \in [1, K_l]$, $l \in [1, L]$. Since during training paths are not fully synchronized, we denote the local copy of the parameters of module (l, e) in path i at iteration t with $\theta(l, e)_i^t$. The collection of parameters used by the i-th path (across all levels) is denoted by θ_i^t. When modules across paths are synchronized, the parameter values of module (l, e) is the same across all paths, and therefore we denote the global copy of the parameters of module (l, e) at iteration t by omitting the path index, as in $\theta(l, e)^t$. Similarly, we denote local and global gradients of module (l, e) with $\Delta(l, e)_i^t$ and $\Delta(l, e)^t$, respectively.

2.4. Coarse Routing

In this section we describe how we route sequences. The basic intuition is to use some context to decide which path is most suitable to process a given sequence. Since we focus on language modeling, we use as context the first 32 tokens of each sequence. At training time, we then use the remaining tokens for learning the model parameters. At test time, we use the remaining tokens to compute perplexity on a held-out validation set. All methods, including dense baselines, will be evaluated in the same way, by calculating perplexity using all but the first 32 tokens of each sequence, which was used to determine the routing decision of our models.

Notice that the outcome of routing is assigning each sequence to a certain path in the network. Collectively, all sequences that are assigned to a particular path form a shard of the original dataset. In this work, there is a one-to-one association between paths and corresponding shards; path π_i is associated to shard $\mathcal{D}_i$.

Next, we describe how we use this prefix of 32 tokens to route, as well as other variants of routing.

2.4.1. Generative routing

In generative routing, the decision about which shard to assign a sequence to is not informed by the task at hand, namely language modeling. Instead, the choice is based on minimizing feature reconstruction error. Given a representation z of the first 32 tokens of a sequence (context), we perform k-Means on the features z of each sequence, and then we use the k-means assign-

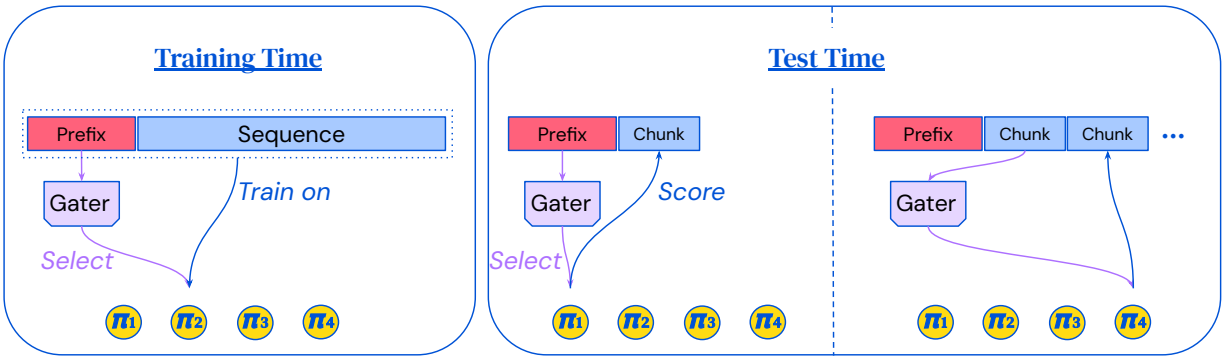

Figure 3 | **routing More Frequently at Test-Time**: At training time (left panel), the router selects the path π_i using the prefix z. We train the chosen path on the whole sequence using the usual language modeling loss. At test time (right panel), the path selected by the router given the prefix is used to score the next chunk of tokens. Then, we re-use the router to choose the most likely path given the new chunk. This process repeats until the whole sequence has been scored.

ment algorithm to shard the data into k shards. If $\{c_1, \ldots, c_k\}$ are the k prototypes learned by k-means, the sequence with prefix z is assigned to shard $\mathcal{D}_{r(z)}$ via:

k-mean assignment: k-means on the embedding and route into the closest cluster

$$r(z) = \arg \min_{i \in [1,k]} ||z - c_i||^2. \quad (1)$$

2.4.2. *Discriminative routing*

Generative routing is agnostic of the language modeling task. In discriminative routing the sharding takes into account how well experts perform on each sequence. The training set is split into two parts. The first and largest part is used to train the paths on data sharded using a generative sharding appproach. The second and much smaller part is used to evaluate paths on each sequence. Let s_i be the log-likelihood score of the path π_i on the sequence associated with prefix z. The index of the path attaining the largest likelihood score, $i^* = \arg\max_j s_j$, is used as label for z. We then train a logistic regression classifier to directly predict i^* from the features z; this trained classifier is used as r. Finally, we use such classifier to re-shard the entire training and validation datasets.

This process is an approximation to Expectation Maximization (Dempster et al., 1977), a coordinate descent method alternating between updating the parameters of the model (paths), and latent variables (path assignments). It is an approximation because, for simplicity, we use as label the top scoring path as opposed to the full posterior distribution over path scores. Of course, this re-assignment process can be repeated for as many times as desired.

We invite those readers interested in diving deeper in the routing mechanism to read subsection 7.2 of the Appendix for further details.

2.4.3. *Routing More Frequently at Test Time*

Routing only once per sequence is critical to distribute training across paths. However, at test time we can afford routing more frequently. Like Figure 3 shows, we can divide a sequence at test time into chunks of $W > 1$ consecutive tokens, e.g., $W = 128$. We can then feed a router with the i-th chunk to predict the best path to use at the $(i + 1)$-th chunk, using a process similar to the one described in subsubsection 2.4.2. The cost of switching paths is small, since the router is run infrequently and in scoring (as opposed to auto-regressive generation) mode. Moreover, only text needs to be communicated to the router and next selected path.

2.4.4. *Overlapping Shards*

Above, each sequence was associated to one and only one shard; in other words, we pre-sharded the dataset into a set of k disjoint partitions.

When the number of shards k is large, the mix-

ture might overfit, particularly when paths share few parameters. To combat this issue we might assign each sequence not just to one shard but to its n "closest" shards. This makes shards overlap with each other, smoothing out the boundaries among themselves, which may improve generalization when the number of shards is relatively large. This can be done *independently* at train time and at evaluation time; that is, one might overlap shards only during train, at eval, or both, or neither. At train time, overlapping the shards limits the specialization of each path, and increases the storage necessary for each shard, but otherwise does not have a computational cost, as each path is still trained independently on its shard. However, overlapping the shards at evaluation means that the model has to be forwarded through multiple paths, which increases the evaluation computational cost.

In this work, when we overlap the shards in training, we use the top-2 choices, while we never overlap the shards at evaluation. We leave to future work other approaches to overlap shards.

2.5. DiLoCo: Review

In this section, we summarize DiLoCo (Douillard et al., 2023) as it lays the foundation of our distributed optimization algorithm, which will be described in the next section. Together with coarse routing of subsection 2.4, DiLoCo is a critical component of DiPaCo.

DiLoCo optimizes a dense model across k workers. First, the original dataset is sharded into k shards, and each shard is associated to a worker. Second, all workers start from the same copy of the model parameters (typically a pretrained model), and perform H (inner) optimization steps over their own shard independently. Third, each worker sends to a central CPU server the difference in parameter space between their new parameters and their initial ones. We refer to these differences as *outer gradients*. The central server averages these outer gradients into a single update vector. Finally, the global parameter vector is updated using an (outer) optimizer, and the new global parameter vector is re-dispatched to all workers for another phase of local training. The process repeats for as many rounds as desired.

In language modeling applications using transformers, the inner and outer optimizers that have been shown to be most effective are respectively AdamW (Kingma and Ba, 2014) and Nesterov momentum (Sutskever et al., 2013). Empirically in prior work, k has been set up to 64 and H up to thousand. Note that other alternatives, such as FedOpt (Reddi et al., 2021), are compatible with this framework.

2.6. DiPaCo

In subsection 2.4, we described methods to pre-shard data, and in subsection 2.5 we described how a model can be trained across several workers that communicated infrequently. This section combines these two approaches towards the efficient training of a composable mixture, a network whose levels are replaced by a mixture of experts. We dub such architecture and training algorithm DiPaCo. In our experiments below, a level is composed of several consecutive transformer blocks.

In the toy illustration of Figure 4 there are three levels, B_1, B_2, and B_3. There is only one module (equivalently, one set of parameters) in B_1, and it is shared across all paths. There are two modules in B_2 (meaning that there will be two distinct parameter vectors for the module at that level); the first is shared by paths π_1 and π_2, and the second by paths π_3 and π_4. Similarly, at the third level π_1 and π_3 share parameters, and π_2 and π_4 share parameters.

The resulting 2×2 DiPaCo has 4 paths in total (as shown on the middle panel of Figure 4). However, the full model need never be fully instantiated, neither during training nor testing. Only paths are realized (as shown on the right panel of Figure 4), and these are trained in parallel with infrequent communication.

At training time DiLoCo is applied at the level of modules, as opposed to the entire network; see lines 11-16 of Algorithm 1. The outer optimizer updates independently the parameters of each module after receiving the corresponding outer gradients from workers processing paths passing through that particular module. For instance and

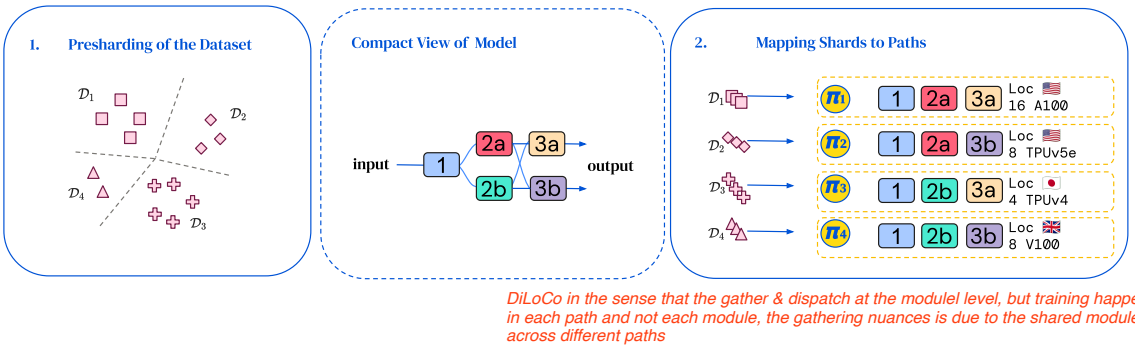

Figure 4 | **DiPaCo**: (left) The dataset is pre-sharded into k shards, $\mathcal{D}_i$ (here $k = 4$). (middle) Compact view of a 2×2 DiPaCo, which is never instantiated. In this toy illustration, there are three levels. Level 2 and 3 have a mixture with two modules each. Level 1 has a single module shared by all paths. (right) We associate each shard $\mathcal{D}_i$ to a *path* π_i, $\forall i \in [1, 4]$. In this toy illustration, a path is the composition of three neural network blocks. The color refers to the id of a module. The figure shows the modular network unrolled across the four paths. These are trained by using DiLoCo which requires communicating only every few hundred steps. In this example, module 2a (in red) is shared by paths π_1 and π_2. Workers associated to paths might use different hardware types (different kind of GPUs or TPUs) and might be placed in far away geographic areas.

with reference to Figure 4, the outer gradient of module 3b is calculated by averaging the gradients of module 3b of the worker processing π_2 and π_4. Such outer gradient is then used to update the parameters of module 3b. A worker does not need to be powerful enough (in terms of memory and compute) to host the entire model, but just a single path. Like in DiLoCo work (Douillard et al., 2023), the inner optimizer is AdamW and the outer optimizer is Nesterov momentum. Similarly, at test time, the paths are instantiated, and served independently, with text routed to each path via a router.

2.6.1. Increasing Capacity

The more paths go through a module, the more opportunities for transfer learning across paths, but also the more constrained learning is and the less capacity the overall mixture has. DiPaCo gives a lot of flexibility in how to set this trade-off. Whether a level is shared by all paths or by a certain subset only does not change the overall framework. The only difference is in the choice of which subset of paths is used when computing the average in line 13 of Algorithm 1.

One extreme choice is to allocate path-specific modules, as shown in Figure 5. In this case, the third level has a mixture with as many modules as there are paths. This is a particularly easy way to increase parameter count in DiPaCo.[1]

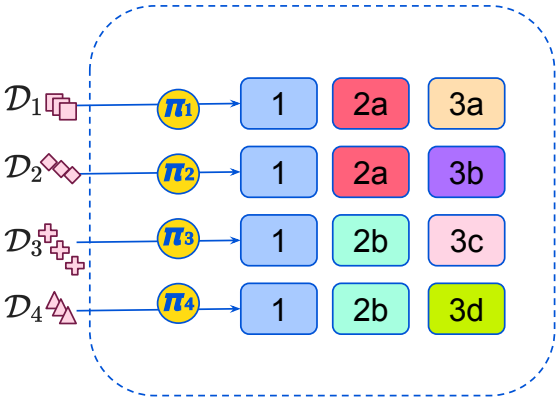

Figure 5 | DiPaCo with more capacity: In this example, level 3 modules are path specific, i.e., modules at that level are not shared by paths.

Clearly, the proposed structure of DiPaCo is merely a first step. We consider it especially ex-

[1]While there is no gradient averaging anymore for these path-specific modules, we still apply the outer optimization of Algorithm 1 because empirically it improves convergence over the default optimizer.

Algorithm 1 DiPaCo (see notation in §2.3)

Require: Num. levels L, num. experts per level K_l, paths π_i with $i \in 1, \dots, P$ and $P = \prod_{i=1}^{L} K_i$. Let $P_{l,e}$ be the number of paths going through module e at level l.
Require: Pre-sharded training set into $\{\mathcal{D}_1, \dots, \mathcal{D}_P\}$ (see §2.4).
Require: Parameters of pretrained model used for initialization: $\bar{\theta}$. $\theta_i^0 = \bar{\theta}, i \in 1, \dots, P$.
Require: Optimizers `InnerOpt` and `OuterOpt`

for `outer step` $t = 1 \dots T$ **do**
  (Optional, in this work done once during training) discriminatively re-shard data (see §2.4.2)
  for `worker` $i = 1 \dots P$ **do**
    $\theta_i^t = \theta_i^{t-1}$
    for `inner step` $n = 1 \dots \tau$ **do**
      $x \sim \mathcal{D}_i$
      $\mathcal{L} \leftarrow f(x, \theta_i^t)$
      $\theta_i^t \leftarrow \texttt{InnerOpt}(\theta_i^t, \nabla_{\mathcal{L}})$
    end for
  end for
  for `level` $l = 1 \dots L$ **do**
    for `expert` $e = 1 \dots K_l$ **do**
      $\Delta(l,e)^t \leftarrow \frac{1}{P_{l,e}} \sum_{i=1}^{P_{l,e}} (\theta(l,e)^{t-1} - \theta(l,e)_i^t)$
      $\theta(l,e)^t \leftarrow \texttt{OuterOpt}(\theta(l,e)^{t-1}, \Delta(l,e)^t)$
    end for
  end for
end for

Inner Optimization: parallel per path

Outer Optimization: parallel per module

citing to co-design the optimization methods and the architecture. In the next section, we discuss some of our initial approaches to adapting optimization methods for our setting.

2.6.2. *Scaling the modular architecture*

DiPaCo is made of multiple levels, each with several expert modules. *e.g* a 16×16 has two levels, which contains 16 modules on each level, producing in total 256 paths. It is easy to scale the architecture size by increasing the number of levels and modules per levels, *e.g* a $32 \times 32 \times 32$ has 32,768 paths. It's difficult to find enough devices for that number of paths, therefore only a subset of the paths can be trained at any moment in time. Potentially, at each start of inner optimization, we could sample a different subset of the paths. Doing so would allow us to scale DiPaCo to arbitrary large size.

2.6.3. *Flat Mixture of Experts*

The extreme form of capacity increase as in subsubsection 2.6.1 would be to have each path be a completely independent network. We will refer to this model as a *flat* mixture of experts (flat MoE). In a flat MoE paths do not share *any* parameter. In the language of subsection 2.3, there is only one level, and $B = B_1$ is the entire network, and $K = K_1$.

Flat MoE is a strong baseline for the compositional version of DiPaCo. In our experiments, we see that it can outperform architectures with shared modules when the number of shards is small compared to the size of the total amount of data.

With generative routing using k-means, this approach is essentially (Gross et al., 2017; Gururangan et al., 2023); in our experiments below we will use discriminative routing as in the compositional models, as this consistently outperforms generative routing.

2.7. Advanced Optimization Techniques

Outer Gradient Norm Rescaling: In DiPaCo, each module can belong to a different number of paths. For instance, in a 16×16 DiPaCo, 16 paths can contribute to a single module, while there could be a level that has path-specific modules which is not shared by any other.

Strangely, rescaling & averaging on the number of contributing paths to a module's gradient is already done in the algorithm, some how empiritcal finding suggests for another scaling with square root of contributing path number?

Consequently, the average outer gradients, $\Delta(l,e)^t$ in Algorithm 1, have significant different norms for different modules. Using the intuition, and also empirical observation, that averaging across a larger number of paths is akin to using a larger batch size, we have rescaled the outer gradient norm by the square root of the number of paths going through a module.

Loss reweighing: In general, shards might have different amounts of data. If training of DiPaCo is distributed by using one worker per shard/path, then according to the uniform average of line 13 in Algorithm 1 smaller shards are over-sampled. To compute an unbiased estimate of the gradient, we therefore weigh the outer gradients proportionally to the shard size:

$$\Delta_{l,e}^{(t)} \leftarrow \sum \alpha_{l,e}(\theta_{l,e}^{(t-1)} - \theta_{l,e}^{(t)}), \tag{2}$$

with:

$$\alpha_{l,e} = \frac{|\mathcal{D}_{l,e}|}{\sum |\mathcal{D}_{l',e'}|}. \tag{3}$$

Early Stopping: By setting aside a small subset of training examples in each shard, we can perform path specific early stopping. In this case, for each path we selected the parameters that yield the lowest loss on the corresponding shard validation set. We found that early stopping improves generalization on small shards, e.g., when the number of shards is large.

3. Infrastructure

We now describe the infrastructure we designed to implement the framework detailed earlier. The infrastructure should achieve the following objectives: 1) reliably train the modular architecture via infrequent synchronization; 2) ensure easy scalability for scenarios involving training across multiple pods or data centers; and 3) provide continuous fault tolerance, allowing every component of the system to quickly recover from failed hosts and preemptions without halting training.

We illustrate the different components in Figure 6. The general training workflow is the following:

1. At the beginning of each training phase, a list of training tasks, each including pairs of path and shard ids, is added to the training task queue maintained by the task scheduler server (in purple).
2. When a training worker in the worker pool (in orange) becomes available, it fetches the next training task from the train task scheduler and performs inner optimization (L5-9 of Algorithm 1) on accelerators. Once the inner optimization is finished, the checkpoint is saved to the Google's distributed file system (Ghemawat et al., 2003) and the training worker becomes available for the next training task. The path to the checkpoint, along with the metadata of the checkpoint (e.g., path ID, outer step ID, etc.), is recorded in a database table (shown in blue). This enables other components to query the checkpoint file path for a given path.
3. When a checkpoint is saved by a training worker (indicated in teal), evaluation tasks for that checkpoint are added to the eval task queue (highlighted in yellow). The eval task queue is then consumed by the evaluation workers in the worker pool to carry out evaluation tasks.
4. The outer optimizer task scheduler (indicated in light blue) distributes outer optimization tasks to sharded outer optimization executors (highlighted in red), each of which is responsible for the outer optimization of a shard of modules (e.g., a single module or a collection of modules). Each executor loads training checkpoints containing the corresponding modules as soon as they appear in the Spanner database table (Corbett et al., 2012) and performs the parameter averaging and outer optimization (L13-14). The checkpoints of the updated module parame-

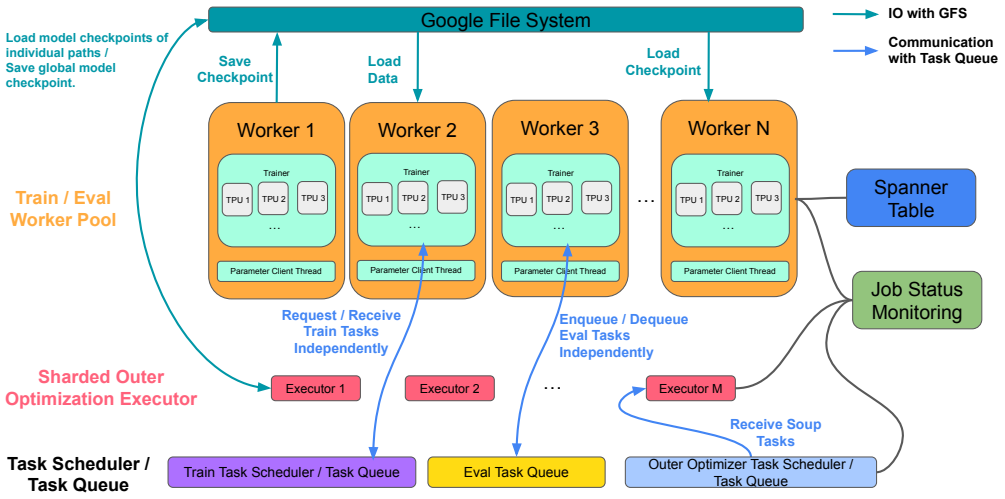

Figure 6 | **Infrastructure**. Training workers (in orange) responsible for performing the inner optimization fetch training task from the train task queue (in purple). Once the training checkpoints are saved to GFS, their paths and metadata (e.g. training step, path ID, phase ID) are written to a Spanner table (in blue) allowing easy look up of checkpoint path given metadata. The evaluation workers and the sharded outer optimization executors (in red) load training checkpoints once they become ready, as signaled by their appearance in the Spanner table. Finally, a monitoring worker (in green) periodically checks the health of workers and reboots them if necessary.

ters are saved to the distributed file system, and their paths with metadata are written to the database table.

5. After completion of all the training tasks, the current training phase concludes, and the subsequent training phase commences and training tasks in the new phase can be started as soon as their corresponding modules finish the outer optimization step. See 3.3 for more detail.
6. Throughout training, a job status monitor (in green) periodically checks the health of workers and task queue servers, and restarts them if they become unresponsive.

All the components are designed to be robust to failures and pre-emptions. We will discuss more details of each infrastructure component in the following subsections.

3.1. Worker Pool

We employ a producer-consumer design pattern to distribute the training tasks. Within each phase, training tasks are added to the task queue, each of which involves training a path for a specific number of steps from a given checkpoint. We maintain a pool of workers, where each worker retrieves training tasks from the task queue iteratively. Each training task is completely independent of other tasks, requiring no synchronization or communication among the workers.

In the event of worker failure or preemption, the fault-tolerant task queue server would return the task from the unavailable worker back to the task queue before reassigning it to another available worker. The task queue server also periodically checkpoints the current task queue, making it possible to recover from server failures or preemptions.

The key advantage of this design is that, thanks to the complete independence among workers, the system can continue making progress even if some workers become unavailable, as long as the worker pool is not empty. It is worth noting that the worker pool can contain *heterogeneous* types of devices across *different regions*, making it easy to scale up further. Additionally, the worker

pool allows for elastic resource utilization by auto-scaling pool size according to resource availability.

3.2. Task Queue System

In practice, the aforementioned task queue system is designed to manage and distribute asynchronous tasks among the workers in the worker pool using remote procedure calls. It consists of three components: the task queue scheduler, the task queue server, and the task queue client. Training tasks are published by the task queue scheduler and sent to the task queue server, while each worker instantiates a task queue client to request tasks from the task queue.

In the multi-host training scenario, where multi-host parallelism is achieved by Single Program, Multiple Data (SPMD), initializing a task queue client on each host would result in each host pulling a different task from the task queue. We solve this issue by implementing a task queue client that synchronizes its results across all hosts. This is achieved by creating a real task queue client only on the first JAX host. All other hosts get the response from the first host by performing a blocking all-gather operation.

We also need to synchronize task queue write operation outside of the main Python thread in some places. For instance, evaluation tasks for a given checkpoint can only be enqueued after checkpointing on all hosts has finished. For this use case, we implemented a barrier, which blocks until each program running on their host have made a call with the same unique key.

3.3. Outer Optimization Efficiency

"scale up number of paths across the world..."

Naïve implementation of outer optimization executor introduces significant overhead as we scale up the number of paths across the world. The following optimizations allow us to scale to hundreds of paths with average time per phase for outer update under 2 minutes (lines 11-16 of Algorithm 1).

Online Parameter Gradient Averaging: As the training tasks within a phase may not finish at the same time, it is possible to reduce the parameter averaging time by loading and accumulating a training checkpoint to the current partial sum as soon as one becomes available. It is also possible to extend to an asynchronous update that doesn't require to wait for all paths before doing an outer update (Liu et al., 2024).

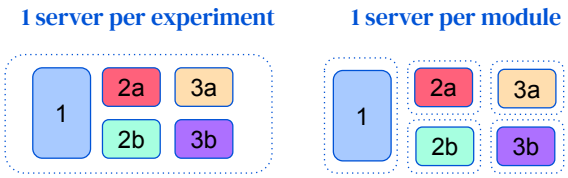

Figure 7 | **Sharded Outer Optimization Executor**: An example of outer optimization executors sharded by module, which significantly reduces the processing time and memory requirement.

Sharded Outer Optimization Executor: As our model architecture is modular, averaging meta-gradients of different modules can be performed independently. By distributing the parameter averaging across multiple servers, as illustrated in Figure 7, we reduce the memory requirement of each averaging task, as well as the overhead of checkpoint loading and averaging. It also allows to reduce the training time as a training task can be started as soon as its corresponding modules have finished their outer update. As a consequence, the overall model is never materialized in a single location but always split across several servers.

Asynchronous Checkpoints Gathering: Each outer optimization executor loads as many checkpoints as the number of paths sharing its associated module. It is possible to load checkpoint from anywhere on earth from the distributed file system (Ghemawat et al., 2003). However, if the checkpoint, located on the server where training took place, is at a certain distance from the outer optimization executor, there is going to be a significant delay. In this case we launch in the background an Effingo process (Google, 2023) to bring the checkpoint to a closer location.

Miscellaneous Improvements: In addition to the previous speed improvements, we also a) cache the parameters of the outer optimizer `OuterOpt`, b) reuse the just-in-time compiled optimizer, and c) load multiple paths in parallel via a multi-threading queue.

3.4. Backup Pool

By default we create as many training workers (see in orange in Figure 6) $\mathcal{W}_i$ as paths/shards of data, $\forall i \in [1, P]$. However as we scale the number of paths, the hardware requirement might become prohibitive. For instance, with 256 paths and 16 `A100` GPUs per worker, we would need 4 096 GPUs. Therefore in practice, the number of workers is often less than the number of paths, and instead, we do multiple *rounds* of training within an outer iteration step until all paths have been trained (L3-10 of Algorithm 1).

To minimize the number of rounds, and therefore maximize speed, we create a backup pool of training workers. Namely, we spawn as many workers as there are paths across multiple accelerator types using a low-tier priority. As soon as an accelerator is available, it is snatched and used for an inner optimization phase. Although device with low-tier priority are preempted frequently, we can benefit from the backup pool since each training task only takes roughly a couple of minutes (the inner optimization L5-9 in Algorithm 1).

Note that this backup pool is particularly useful when used with the path sampling idea mentioned in subsubsection 2.6.2.

4. Experiments

We consider a language modeling task on the C4 dataset, derived from Common Crawl (Raffel et al., 2020), tokenized with a SentencePiece tokenizer (Kudo and Richardson, 2018) with a vocabulary size of 32,000. We report perplexity on the validation set against number of weight update steps used at training time, which is a close proxy for wall-clock time if all computations are done on the same accelerator type. The total number of weight update steps is set to 88,000. All paths on DiPaCo have size 150 million parameters, using a transformer with 12 blocks, 896 dimensional hidden states and 16 heads.

We build modular networks of varying number of paths, and varying levels of module reuse. We compare to the performance of a dense transformer language model of size 1.3B, which has 24 blocks, 2048 dimensional hidden states and with the same number of heads; and a dense model that is the size of one DiPaCo path. These models are also trained for 88,000 weight update steps.

We warn the reader that this comparison is *not* standard in the literature, as weight updates for DiPaCo see more tokens and use more FLOPs when the number of paths is larger. Nevertheless, while our modular networks is made of hundreds of paths, they are all trained in parallel, and thus our training wallclock time is 45% less than the 1.3B dense counterpart, and roughly equivalent to the 150M parameter single-path counterpart.

In our experiments we have searched over relatively few hyper-parameters: mainly learning rate and value of Nesterov momentum. We use a sequence length of 1,024 tokens and a batch size of 512. At eval time, we consider sequences of 2,048 tokens. We use cosine learning rate scheduling with peak value of 0.0004. We list all the hyper-parameters in the appendix (subsection 7.1).

All instances of DiPaCo use one phase of discriminative routing (see 2.4.2 and 7.2.1). The $16 \times 16 = 256$ path DiPaCo uses top-2 overlapping shards (2.4.4) at training time (this does not make training slower in wallclock or FLOPs, although it does increase the size of the shards).

4.1. Comparison with 1B dense model

Figure 8 compares the convergence curve of the 1.3B dense language model with the 16×16 DiPaCo. DiPaCo with 256 paths of size 150M partially sharing parameters, and one routing decision per document, is nearly sufficient to match the performance of a dense model of size 1.3B. The remaining performance gap with the dense 1.3B model can be closed with early stopping and routing every 64 tokens at test time, see Table 3.

We again warn the reader that the x axis in Figure 8 is not FLOP equivalent for the different

Ok, this is more interesting as it shows test time multiple gating helps DIPaCo trained MoE to reach performance of a baseline Full model, (competing with sharded baseline model means nothing)

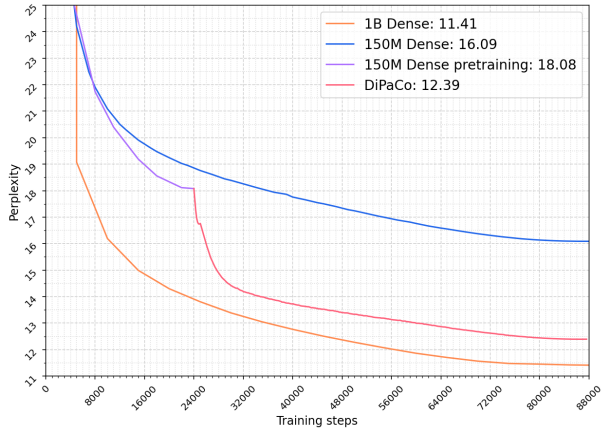

Figure 8 | **Convergence curves of DiPaCo vs Dense Baseline**: We first pretrain a 150M parameters model for 24k training steps (purple). We then finetune a 16x16 DiPaCo $P = 256$ (red). The remaining gap to the dense 1.3B parameters baseline (orange) is reached by gating more frequently at test time.

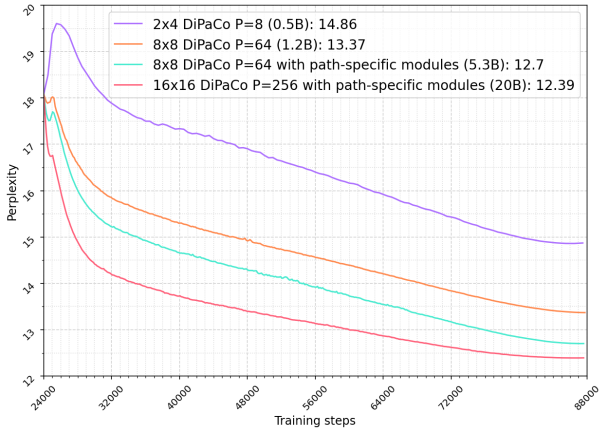

Figure 9 | **Scaling number of paths in DiPaCo**: We report validation perplexity of DiPaCo with different numbers of paths and parameters. We increase capacity by varying both the number of paths (from 8 to 256) and by using path-specific modules. The size of a path (roughly equivalent to serving cost during deployment) is 150M parameters for all models in this figure.

models. For the 150M baseline model and for DiPaCo, the x axis roughly corresponds to equivalent wall-clock (and the 1.3B model is slower w.r.t. wall-clock for the same number of updates). On the other hand, DiPaCo uses many more FLOPs and sees more tokens per training step than the dense baselines. However, it is not trivial to parallelize training of the dense models to the scale of DiPaCo, see the comparisons in subsection 4.3 and in particular Table 1, and in (Douillard et al., 2023).

4.2. Number of paths and parameters

Figure 9 shows that generalization improves as more paths and more overall parameters are added to the mixture. Since the overhead of DiLoCo is minimal, the x axis could be replaced with wall-clock time. Of course, 16×16 DiPaCo requires 32 times more compute than 2×4 DiPaCo, but these additional paths can be trained in parallel with very little communication. In this experiment the inner optimization consists of $\tau = 150$ steps. For all experiments with *path-specific modules*, the transformer blocks 0, 5, 6 11, and the embedding matrix are not communicated across paths.

4.3. Parameter sharing between paths

At one end of the spectrum, as described in subsubsection 2.6.3, we use paths that share no parameters, and can be trained fully independently. Continuing the trend from subsection 4.2, where there is monotone improvement in validation performance as parameters are added, we see improvements by unsharing all parameters for small number of paths. Using 64 unshared paths we can approach a 16×16 DiPaCo that has no unsharing.

However, this model starts over-fitting once the number of shards (and corresponding experts) is further increased as shown in Table 2. We are able to recover some performance by overlapping shards, but still we find that we cannot naively scale the number of fully independent paths. We consider approaches for more sophisticated overlapping of shards and path branching to be a promising direction for future work.

At the other end of the spectrum, paths can share all parameters, effectively collapsing back into a single path and recovering the performance of the original DiLoCo (Douillard et al., 2023). We can see that the base model trained with DiLoCo, or base model overtrained for many steps,

Model	Time	Compute and Data	Total Parameters	Validation PPL
Baseline	1×	1×	150M	16.23
DiLoCo $P = 8$	1×	8×	150M	15.02
DiLoCo $P = 64$	1×	64×	150M	14.96
Flat MoE $P = 8$	1×	8×	1.2B	14.62
Flat MoE $P = 64$	1×	64×	9.6B	12.76
DiPaCo 2×4	1×	8×	0.5B	14.86
DiPaCo 8×8	1×	64×	1.2B	13.37
DiPaCo 8×8+ Path Specific Modules	1×	64×	5.3B	**12.70**
Baseline, 8× steps	8×	8×	150M	14.72

Table 1 | **DiPaCo vs. Flat MoE vs. DiLoCo**: We compare DiLoCo and Flat MoE (2.6.3) vs. DiPaCo. We also compare against the baseline trained for the same FLOPs as the distributed models. In DiLoCo all paths are collapsed together at each outer optimization step. In Flat MoE, all paths are independent. In DiPaCo, only a subset of the parameters are "collapsed" using information from a subset of paths. Because of the distributed framework, DiPaCo is trained in the same wallclock time as DiLoCo, which is roughly the wallclock of the baseline.

although receiving the same number of FLOPs and tokens as the larger DiPaCo models, do not have the capacity to make use of these extra FLOPs. Of course a properly sized baseline would do much better; and indeed the 1.3B model above already has better perplexity. However, DiPaCo consists of paths that each require only 150M size island of compute both in train and testing, and its distributed training algorithm is built into the architecture, so that only paths of 150M parameters ever need to be materialized on any training or deployment worker. We compare all these variants in Table 1.

4.4. Routing frequency during evaluation

We demonstrate the results of routing more frequently at test time in Table 3. We observe a significant improvement of 0.74 perplexity points going from routing once per sequence of 1,024 tokens to every 128 tokens. Further increasing the granularity of the re-routing consistently improve results, although more marginally so. This shows that at training time we can route at the sequence level to enable pre-sharding for more efficient learning, and then at evaluation time we can gain back performance by routing more frequently. By routing tokens potentially to a different path every 64 tokens, DiPaCo reaches 11.38 perplexity, matching the performance of a 1B dense model (11.41). Since at any time DiPaCo uses a single path of size 150M parameters, it matches the performance of the 1B model using 6 times less parameters, and therefore, requiring less compute at inference time.

# fully independent paths	Validation PPL
$P = 8$	14.6
$P = 16$	13.9
$P = 256$	14.2

Table 2 | **Flat MoE (independent paths) overfits as number of paths increase**: Validation perplexity of DiPaCo with no shared modules, as described in subsubsection 2.6.3 after 10K steps of training. With overlapping shards as described in subsubsection 2.4.4 and early stopping, we can improve the $P = 256$ result to 13.6, but still the models overfit at 64K steps with 256 paths. In contrast, there is no overfitting at 256 paths with overlapping shards with a 16×16 DiPaCo.

better performance on test set observed when the number of independent paths is at ~16

Early Stopping	Route Every	Perplexity
✗	Once per sequence	12.39
✓		12.22
✓	128	11.48
	64	**11.38**
	32	11.31
	16	11.26

Table 3 | **Frequent Routing at Eval-Time**: Validation perplexity of a 16×16 DiPaCo with $P = 256$. Despite using paths of size 150M ($> 6\times$ fewer parameters), we match the performance of a dense 1B model (11.41) by potentially re-routing to a different path every 64 tokens, although the model may choose to route to the exact same path that it selected previously.

4.5. Parameter synchronization with DiLoCo vs. true gradients

Finally, we ablate the effect of using the partially synchronous optimization DiLoCo to understand whether we lose performance by communicating less frequently at training time. In the setting where all devices are actually collocated and there is no constraint in terms of communication, we can train the DiPaCo model in a fully synchronous manner without using DiLoCo: At every step, each path computes gradients on **its own batch of data** from its own data shard; Gradients across all paths are then exchanged and aggregated module by module; Finally, the model performs one step of AdamW update with the aggregated gradient. To our surprise, we did not observe a significant performance gap between the fully synchronous training setting and the partially synchronous training setting. More specifically, DiPaCo trained with DiLoCo slightly outperforms their fully-synchronously-trained version by 0.3 and 0.6 perplexity points when using a 2×2 and 4×4 architecture, respectively. At 8×8 DiPaCo trained fully synchronously reaches better perplexity by only 0.1 perplexity despite communicating hundred of times more. This suggests that DiLoCo is an effective distributed optimization algorithm for DiPaCo.

5. Related Work

As mentioned in the introduction, this work shares the same motivation and intuitions expressed in Pathways (Dean, 2021). Unlike the Pathways framework (Barham et al., 2022) which supports training of general modular multimodal multitask asynchronous systems, we propose a particular instantiation of a modular system that supports such kind of distributed training. Our approach also shares motivations and intuitions with Borzunov et al. (2022); Ryabinin and Gusev (2020). The key difference in this work is that each worker trains a *path* through modules, rather than a module.

5.1. Modularity

There is a large body of literature on modularity, although this is not always framed in such terms. At one end of the spectrum there are approaches that use a very large number of very tiny modules. For instance, RETRO (Borgeaud et al., 2021) is one such approach, whereby there is one module per token n-gram of a large retrieval dataset, and the modules are merely vectors of biases. At the other end of the spectrum there are approaches that use very few (e.g., two) but very large modules. For instance, Flamingo (Alayrac et al., 2022) is a vision language model composed by two main modules, a pre-trained visual encoder and a pre-trained language model. Similarly, Dalmia et al. (2023) proposed another approach where modules are pre-trained language models and speech recognition encoders. The authors showed how such modules could be plugged in different ways to enable 0-shot learning of entirely new tasks, such as producing a speech recognizer that translates into another language.

These lines of work are representative of the vast majority of the literature on modularity. There is however also some work where the number of modules and their size is intermediate. For instance, Purushwalkam et al. (2019) showed how softly gated modules could be used in visual question answering to handle compositional tasks. In multi-task cross-lingual transfer, Pfeiffer et al. (2020) used instead a network where modules are very small adapters, and the vast majority of

parameters are shared across paths. We refer the reader to the survey by Pfeiffer et al. (2023) for additional pointers, and focus next on modularity via mixture models.

5.2. Mixture of Experts

Together with ensembling and boosting, mixture of experts (MoE) are an early approach to modularity in the machine learning literature (Jacobs et al., 1991; Jordan and Jacobs, 1994). Unlike ensembling, mixture models could be more efficient if an input is processed only by a few experts. Unlike boosting, training of mixture models is efficient because it can be parallelized, while in boosting the process is inherently sequential.

While early work considered the problem of hierarchical gating of flat mixtures (Jordan and Jacobs, 1994), mixtures at multiple levels appeared only more recently. For instance, Eigen et al. (2014) proposed a two-level mixture where paths were softly mixed and trained jointly via stochastic gradient descent. In that paper, the authors recognized the difficulty of learning non-degenerate routing functions and discussed some workarounds. The issue of how to learn a good routing function has been a major topic of research in the field since then.

In their seminal work, Shazeer et al. (2017) proposed a very large mixture of experts LSTM model for sequence modeling tasks. Most works MoE for sequence-modeling works that followed (Artetxe et al., 2021; Clark et al., 2022; Fedus et al., 2021; Lepikhin et al., 2021) have used a recipe whereby FFN layers of transformers are replaced by mixtures. These MoEs operate at the token level, and training operates similarly to dense model training: all paths are synchronized at every gradient descent step. Token-MoE are currently state of the art with respect to training FLOP efficiency, but require even more co-located accelerators than the equivalent-activated dense model for training.

In contrast, (Gururangan et al., 2023) trains experts independently using a document level router; this approach had been used in computer vision by Gross et al. (2017), and it also appeared in the federated learning literature (Reisser et al., 2021). Our flat MoE baseline is closely related to the approach in (Gururangan et al., 2023), except we used a discriminative router as opposed to k-Means, and we used the first 32 tokens of a document (and the features from the base Transformer LM) as the input to the router, as opposed to SVD of tf-idf vectors.

Finally, the idea to iteratively shard the data by domain to train models in parallel, followed by a model merging phase has also been proposed by Li et al. (2022). In this work, we take this approach a step further by learning a mixture of experts model, and by automatically (and even discriminatively) sharding the data.

Iteratively shard the data by domain to train models in parallel, followed by a model merging phase (prev idea) II learning a discriminative data-sharding (actually no, it's not so much of a different that this previous work in fact)

6. Limitations

The most salient limitation to DiPaCo is with respect to FLOP efficiency. In this work we made no effort to optimize its FLOP efficiency; and in the results presented, DiPaCo is significantly less FLOP efficient per evaluation perplexity than a standard dense compute optimal model.

This work also does not study scaling laws of DiPaCo. We work at a single path-size scale, and on one dataset.

We also made no effort in this work to optimize deployment of DiPaCo. In particular, if we were to naively route more frequently during deployment as in Section (2.4.3), we would need to re-compute the KV-cache for each query after each re-route.

The severity of these limitations depends on the constraints of the training and deployment infrastructure, as well as on the task at hand. For instance, they may matter less when the computing infrastructure consists of hundreds of small and poorly connected islands of devices for tasks where sequences can be mostly associated to one (or very few) domains. In general, addressing these limitations constitutes avenue of future research.

7. Conclusions and future work

In this work, we architect an ML model as a union of modules, and we define input-output mappings

that are paths through these modules. We show that we can train paths (almost) independently, and (virtually) recombine these paths back into the desired large model. In particular, we were able to show that distributed training by path can match the performance of a 1B dense language model for a similar wall clock time.

The design of the algorithm and architecture is guided by practical trade-offs. In our work we assume that communication is costly but compute is not. In other words, the compute that counts the most is the one of co-located devices that host a path. As long as communication between such relatively small islands of devices is limited, we can scale by distributing computation across a large number of such islands.

We introduced two key ingredients to operate under these constraints: 1) we route coarsely at the sequence level and pre-shard data by path, and 2) we extend DiLoCo to update parameters of modules shared by multiple paths. We believe that by combining these two ideas we have made a step towards a potentially much more scalable paradigm to train large-models.

There are several avenues of future work. In our investigations we made no effort to control the FLOP efficiency of DiPaCo, focusing on wall-clock efficiency. Naturally, there are several straightforward design choices that could increase FLOP efficiency, for example allowing some paths to co-locate. In addition, our approaches to sharding are fairly simple, and there are many interesting possibilities for more sophisticated sharding. DiPaCo was designed with continual learning in mind, and we would like to apply it in that setting eventually. Last but certainly not least, scaling up is of utmost interest. Shared with Dean (2021); Raffel (2023a), our long-term dream is to further refine this approach and produce a never-ending, community-driven, modular learning system that can be used by everyone to compose new predictors out of existing modules, and thus efficiently develop entirely new models and capabilities in a positive feedback loop.

References

J.-B. Alayrac, J. Donahue, P. Luc, A. Miech, I. Barr, Y. Hasson, K. Lenc, A. Mensch, K. Millican, M. Reynolds, R. Ring, E. Rutherford, S. Cabi, T. Han, Z. Gong, S. Samangooei, M. Monteiro, J. Menick, S. Borgeaud, A. Brock, A. Nematzadeh, S. Sharifzadeh, M. Binkowski, R. Barreira, O. Vinyals, A. Zisserman, and K. Simonyan. Flamingo: a visual language model for few-shot learning. In *NeurIPS*, 2022.

R. Anil, S. Borgeaud, Y. Wu, J.-B. Alayrac, J. Yu, R. Soricut, J. Schalkwyk, A. M. Dai, A. Hauth, K. Millican, D. Silver, S. Petrov, M. Johnson, I. Antonoglou, J. Schrittwieser, A. Glaese, J. Chen, E. Pitler, T. Lillicrap, A. Lazaridou, O. Firat, J. Molloy, M. Isard, P. R. Barham, T. Hennigan, B. Lee, F. Viola, M. Reynolds, Y. Xu, R. Doherty, E. Collins, C. Meyer, E. Rutherford, E. Moreira, K. Ayoub, M. Goel, G. Tucker, E. Piqueras, M. Krikun, I. Barr, N. Savinov, I. Danihelka, B. Roelofs, A. White, A. Andreassen, T. von Glehn, L. Yagati, M. Kazemi, L. Gonzalez, M. Khalman, J. Sygnowski, A. Frechette, C. Smith, L. Culp, L. Proleev, Y. Luan, X. Chen, J. Lottes, N. Schucher, F. Lebron, A. Rrustemi, N. Clay, P. Crone, T. Kocisky, J. Zhao, B. Perz, D. Yu, H. Howard, A. Bloniarz, J. W. Rae, H. Lu, L. Sifre, M. Maggioni, F. Alcober, D. Garrette, M. Barnes, S. Thakoor, J. Austin, G. Barth-Maron, W. Wong, R. Joshi, R. Chaabouni, D. Fatiha, A. Ahuja, R. Liu, Y. Li, S. Cogan, J. Chen, C. Jia, C. Gu, Q. Zhang, J. Grimstad, A. J. Hartman, M. Chadwick, G. S. Tomar, X. Garcia, E. Senter, E. Taropa, T. S. Pillai, J. Devlin, M. Laskin, D. de Las Casas, D. Valter, C. Tao, L. Blanco, A. P. Badia, D. Reitter, M. Chen, J. Brennan, C. Rivera, S. Brin, S. Iqbal, G. Surita, J. Labanowski, A. Rao, S. Winkler, E. Parisotto, Y. Gu, K. Olszewska, Y. Zhang, R. Addanki, A. Miech, A. Louis, L. E. Shafey, D. Teplyashin, G. Brown, E. Catt, N. Attaluri, J. Balaguer, J. Xiang, P. Wang, Z. Ashwood, A. Briukhov, A. Webson, S. Ganapathy, S. Sanghavi, A. Kannan, M.-W. Chang, A. Stjerngren, J. Djolonga, Y. Sun, A. Bapna, M. Aitchison, P. Pejman, H. Michalewski, T. Yu, C. Wang, J. Love, J. Ahn, D. Bloxwich, K. Han,

P. Humphreys, T. Sellam, J. Bradbury, V. Godbole, S. Samangooei, B. Damoc, A. Kaskasoli, S. M. R. Arnold, V. Vasudevan, S. Agrawal, J. Riesa, D. Lepikhin, R. Tanburn, S. Srinivasan, H. Lim, S. Hodkinson, P. Shyam, J. Ferret, S. Hand, A. Garg, T. L. Paine, J. Li, Y. Li, M. Giang, A. Neitz, Z. Abbas, S. York, M. Reid, E. Cole, A. Chowdhery, D. Das, D. Rogozińska, V. Nikolaev, P. Sprechmann, Z. Nado, L. Zilka, F. Prost, L. He, M. Monteiro, G. Mishra, C. Welty, J. Newlan, D. Jia, M. Allamanis, C. H. Hu, R. de Liedekerke, J. Gilmer, C. Saroufim, S. Rijhwani, S. Hou, D. Shrivastava, A. Baddepudi, A. Goldin, A. Ozturel, A. Cassirer, Y. Xu, D. Sohn, D. Sachan, R. K. Amplayo, C. Swanson, D. Petrova, S. Narayan, A. Guez, S. Brahma, J. Landon, M. Patel, R. Zhao, K. Villela, L. Wang, W. Jia, M. Rahtz, M. Giménez, L. Yeung, H. Lin, J. Keeling, P. Georgiev, D. Mincu, B. Wu, S. Haykal, R. Saputro, K. Vodrahalli, J. Qin, Z. Cankara, A. Sharma, N. Fernando, W. Hawkins, B. Neyshabur, S. Kim, A. Hutter, P. Agrawal, A. Castro-Ros, G. van den Driessche, T. Wang, F. Yang, S. yiin Chang, P. Komarek, R. McIlroy, M. Lučić, G. Zhang, W. Farhan, M. Sharman, P. Natsev, P. Michel, Y. Cheng, Y. Bansal, S. Qiao, K. Cao, S. Shakeri, C. Butterfield, J. Chung, P. K. Rubenstein, S. Agrawal, A. Mensch, K. Soparkar, K. Lenc, T. Chung, A. Pope, L. Maggiore, J. Kay, P. Jhakra, S. Wang, J. Maynez, M. Phuong, T. Tobin, A. Tacchetti, M. Trebacz, K. Robinson, Y. Katariya, S. Riedel, P. Bailey, K. Xiao, N. Ghelani, L. Aroyo, A. Slone, N. Houlsby, X. Xiong, Z. Yang, E. Gribovskaya, J. Adler, M. Wirth, L. Lee, M. Li, T. Kagohara, J. Pavagadhi, S. Bridgers, A. Bortsova, S. Ghemawat, Z. Ahmed, T. Liu, R. Powell, V. Bolina, M. Iinuma, P. Zablotskaia, J. Besley, D.-W. Chung, T. Dozat, R. Comanescu, X. Si, J. Greer, G. Su, M. Polacek, R. L. Kaufman, S. Tokumine, H. Hu, E. Buchatskaya, Y. Miao, M. Elhawaty, A. Siddhant, N. Tomasev, J. Xing, C. Greer, H. Miller, S. Ashraf, A. Roy, Z. Zhang, A. Ma, A. Filos, M. Besta, R. Blevins, T. Klimenko, C.-K. Yeh, S. Changpinyo, J. Mu, O. Chang, M. Pajarskas, C. Muir, V. Cohen, C. L. Lan, K. Haridasan, A. Marathe, S. Hansen, S. Douglas, R. Samuel, M. Wang, S. Austin, C. Lan, J. Jiang, J. Chiu, J. A. Lorenzo, L. L. Sjösund, S. Cevey, Z. Gleicher, T. Avrahami, A. Boral, H. Srinivasan, V. Selo, R. May, K. Aisopos, L. Hussenot, L. B. Soares, K. Baumli, M. B. Chang, A. Recasens, B. Caine, A. Pritzel, F. Pavetic, F. Pardo, A. Gergely, J. Frye, V. Ramasesh, D. Horgan, K. Badola, N. Kassner, S. Roy, E. Dyer, V. Campos, A. Tomala, Y. Tang, D. E. Badawy, E. White, B. Mustafa, O. Lang, A. Jindal, S. Vikram, Z. Gong, S. Caelles, R. Hemsley, G. Thornton, F. Feng, W. Stokowiec, C. Zheng, P. Thacker, Çağlar Ünlü, Z. Zhang, M. Saleh, J. Svensson, M. Bileschi, P. Patil, A. Anand, R. Ring, K. Tsihlas, A. Vezer, M. Selvi, T. Shevlane, M. Rodriguez, T. Kwiatkowski, S. Daruki, K. Rong, A. Dafoe, N. FitzGerald, K. Gu-Lemberg, M. Khan, L. A. Hendricks, M. Pellat, V. Feinberg, J. Cobon-Kerr, T. Sainath, M. Rauh, S. H. Hashemi, R. Ives, Y. Hasson, Y. Li, E. Noland, Y. Cao, N. Byrd, L. Hou, Q. Wang, T. Sottiaux, M. Paganini, J.-B. Lespiau, A. Moufarek, S. Hassan, K. Shivakumar, J. van Amersfoort, A. Mandhane, P. Joshi, A. Goyal, M. Tung, A. Brock, H. Sheahan, V. Misra, C. Li, N. Rakićević, M. Dehghani, F. Liu, S. Mittal, J. Oh, S. Noury, E. Sezener, F. Huot, M. Lamm, N. D. Cao, C. Chen, G. Elsayed, E. Chi, M. Mahdieh, I. Tenney, N. Hua, I. Petrychenko, P. Kane, D. Scandinaro, R. Jain, J. Uesato, R. Datta, A. Sadovsky, O. Bunyan, D. Rabiej, S. Wu, J. Zhang, G. Vasudevan, E. Leurent, M. Alnahlawi, I. Georgescu, N. Wei, I. Zheng, B. Chan, P. G. Rabinovitch, P. Stanczyk, Y. Zhang, D. Steiner, S. Naskar, M. Azzam, M. Johnson, A. Paszke, C.-C. Chiu, J. S. Elias, A. Mohiuddin, F. Muhammad, J. Miao, A. Lee, N. Vieillard, S. Potluri, J. Park, E. Davoodi, J. Zhang, J. Stanway, D. Garmon, A. Karmarkar, Z. Dong, J. Lee, A. Kumar, L. Zhou, J. Evens, W. Isaac, Z. Chen, J. Jia, A. Levskaya, Z. Zhu, C. Gorgolewski, P. Grabowski, Y. Mao, A. Magni, K. Yao, J. Snaider, N. Casagrande, P. Suganthan, E. Palmer, G. Irving, E. Loper, M. Faruqui, I. Arkatkar, N. Chen, I. Shafran, M. Fink, A. Castaño, I. Giannoumis, W. Kim, M. Rybiński, A. Sreevatsa, J. Prendki, D. Soergel, A. Goedeckemeyer, W. Gierke, M. Jafari, M. Gaba, J. Wiesner, D. G. Wright, Y. Wei, H. Vashisht, Y. Kulizhskaya, J. Hoover, M. Le, L. Li, C. Iwuanyanwu, L. Liu, K. Ramirez, A. Khorlin, A. Cui, T. LIN,

M. Georgiev, M. Wu, R. Aguilar, K. Pallo, A. Chakladar, A. Repina, X. Wu, T. van der Weide, P. Ponnapalli, C. Kaplan, J. Simsa, S. Li, O. Dousse, F. Yang, J. Piper, N. Ie, M. Lui, R. Pasumarthi, N. Lintz, A. Vijayakumar, L. N. Thiet, D. Andor, P. Valenzuela, C. Paduraru, D. Peng, K. Lee, S. Zhang, S. Greene, D. D. Nguyen, P. Kurylowicz, S. Velury, S. Krause, C. Hardin, L. Dixon, L. Janzer, K. Choo, Z. Feng, B. Zhang, A. Singhal, T. Latkar, M. Zhang, Q. Le, E. A. Abellan, D. Du, D. McKinnon, N. Antropova, T. Bolukbasi, O. Keller, D. Reid, D. Finchelstein, M. A. Raad, R. Crocker, P. Hawkins, R. Dadashi, C. Gaffney, S. Lall, K. Franko, E. Filonov, A. Bulanova, R. Leblond, V. Yadav, S. Chung, H. Askham, L. C. Cobo, K. Xu, F. Fischer, J. Xu, C. Sorokin, C. Alberti, C.-C. Lin, C. Evans, H. Zhou, A. Dimitriev, H. Forbes, D. Banarse, Z. Tung, J. Liu, M. Omernick, C. Bishop, C. Kumar, R. Sterneck, R. Foley, R. Jain, S. Mishra, J. Xia, T. Bos, G. Cideron, E. Amid, F. Piccinno, X. Wang, P. Banzal, P. Gurita, H. Noga, P. Shah, D. J. Mankowitz, A. Polozov, N. Kushman, V. Krakovna, S. Brown, M. Bateni, D. Duan, V. Firoiu, M. Thotakuri, T. Natan, A. Mohananey, M. Geist, S. Mudgal, S. Girgin, H. Li, J. Ye, O. Roval, R. Tojo, M. Kwong, J. Lee-Thorp, C. Yew, Q. Yuan, S. Bagri, D. Sinopalnikov, S. Ramos, J. Mellor, A. Sharma, A. Severyn, J. Lai, K. Wu, H.-T. Cheng, D. Miller, N. Sonnerat, D. Vnukov, R. Greig, J. Beattie, E. Caveness, L. Bai, J. Eisenschlos, A. Korchemniy, T. Tsai, M. Jasarevic, W. Kong, P. Dao, Z. Zheng, F. Liu, F. Yang, R. Zhu, M. Geller, T. H. Teh, J. Sanmiya, E. Gladchenko, N. Trdin, A. Sozanschi, D. Toyama, E. Rosen, S. Tavakkol, L. Xue, C. Elkind, O. Woodman, J. Carpenter, G. Papamakarios, R. Kemp, S. Kafle, T. Grunina, R. Sinha, A. Talbert, A. Goyal, D. Wu, D. Owusu-Afriyie, C. Du, C. Thornton, J. Pont-Tuset, P. Narayana, J. Li, S. Fatehi, J. Wieting, O. Ajmeri, B. Uria, T. Zhu, Y. Ko, L. Knight, A. Héliou, N. Niu, S. Gu, C. Pang, D. Tran, Y. Li, N. Levine, A. Stolovich, N. Kalb, R. Santamaria-Fernandez, S. Goenka, W. Yustalim, R. Strudel, A. Elqursh, B. Lakshminarayanan, C. Deck, S. Upadhyay, H. Lee, M. Dusenberry, Z. Li, X. Wang, K. Levin, R. Hoffmann, D. Holtmann-Rice, O. Bachem, S. Yue, S. Arora, E. Malmi, D. Mirylenka, Q. Tan, C. Koh, S. H. Yeganeh, S. Põder, S. Zheng, F. Pongetti, M. Tariq, Y. Sun, L. Ionita, M. Seyedhosseini, P. Tafti, R. Kotikalapudi, Z. Liu, A. Gulati, J. Liu, X. Ye, B. Chrzaszcz, L. Wang, N. Sethi, T. Li, B. Brown, S. Singh, W. Fan, A. Parisi, J. Stanton, C. Kuang, V. Koverkathu, C. A. Choquette-Choo, Y. Li, T. Lu, A. Ittycheriah, P. Shroff, P. Sun, M. Varadarajan, S. Bahargam, R. Willoughby, D. Gaddy, I. Dasgupta, G. Desjardins, M. Cornero, B. Robenek, B. Mittal, B. Albrecht, A. Shenoy, F. Moiseev, H. Jacobsson, A. Ghaffarkhah, M. Rivière, A. Walton, C. Crepy, A. Parrish, Y. Liu, Z. Zhou, C. Farabet, C. Radebaugh, P. Srinivasan, C. van der Salm, A. Fidjeland, S. Scellato, E. Latorre-Chimoto, H. Klimczak-Plucińska, D. Bridson, D. de Cesare, T. Hudson, P. Mendolicchio, L. Walker, A. Morris, I. Penchev, M. Mauger, A. Guseynov, A. Reid, S. Odoom, L. Loher, V. Cotruta, M. Yenugula, D. Grewe, A. Petrushkina, T. Duerig, A. Sanchez, S. Yadlowsky, A. Shen, A. Globerson, A. Kurzrok, L. Webb, S. Dua, D. Li, P. Lahoti, S. Bhupatiraju, D. Hurt, H. Qureshi, A. Agarwal, T. Shani, M. Eyal, A. Khare, S. R. Belle, L. Wang, C. Tekur, M. S. Kale, J. Wei, R. Sang, B. Saeta, T. Liechty, Y. Sun, Y. Zhao, S. Lee, P. Nayak, D. Fritz, M. R. Vuyyuru, J. Aslanides, N. Vyas, M. Wicke, X. Ma, T. Bilal, E. Eltyshev, D. Balle, N. Martin, H. Cate, J. Manyika, K. Amiri, Y. Kim, X. Xiong, K. Kang, F. Luisier, N. Tripuraneni, D. Madras, M. Guo, A. Waters, O. Wang, J. Ainslie, J. Baldridge, H. Zhang, G. Pruthi, J. Bauer, F. Yang, R. Mansour, J. Gelman, Y. Xu, G. Polovets, J. Liu, H. Cai, W. Chen, X. Sheng, E. Xue, S. Ozair, A. Yu, C. Angermueller, X. Li, W. Wang, J. Wiesinger, E. Koukoumidis, Y. Tian, A. Iyer, M. Gurumurthy, M. Goldenson, P. Shah, M. Blake, H. Yu, A. Urbanowicz, J. Palomaki, C. Fernando, K. Brooks, K. Durden, H. Mehta, N. Momchev, E. Rahimtoroghi, M. Georgaki, A. Raul, S. Ruder, M. Redshaw, J. Lee, K. Jalan, D. Li, G. Perng, B. Hechtman, P. Schuh, M. Nasr, M. Chen, K. Milan, V. Mikulik, T. Strohman, J. Franco, T. Green, D. Hassabis, K. Kavukcuoglu, J. Dean, and O. Vinyals. Gemini: A family of highly capable multimodal models, 2023.

M. Artetxe, S. Bhosale, N. Goyal, T. Mihaylov, M. Ott, S. Shleifer, X. V. Lin, J. Du, S. Iyer, R. Pasunuru, et al. Efficient large scale language modeling with mixtures of experts. *arXiv preprint arXiv:2112.10684*, 2021.

P. Barham, A. Chowdhery, J. Dean, S. Ghemawat, S. Hand, D. Hurt, M. Isard, H. Lim, R. Pang, S. Roy, B. Saeta, P. Schuh, R. Sepassi, L. E. Shafey, C. A. Thekkath, and Y. Wu. Pathways: Asynchronous distributed dataflow for ml, 2022.

S. Borgeaud, A. Mensch, J. Hoffmann, T. Cai, E. Rutherford, K. Millican, G. van den Driessche, J. Lespiau, B. Damoc, A. Clark, D. de Las Casas, A. Guy, J. Menick, R. Ring, T. Hennigan, S. Huang, L. Maggiore, C. Jones, A. Cassirer, A. Brock, M. Paganini, G. Irving, O. Vinyals, S. Osindero, K. Simonyan, J. W. Rae, E. Elsen, and L. Sifre. Improving language models by retrieving from trillions of tokens, 2021. URL https://arxiv.org/abs/2112.04426.

A. Borzunov, D. Baranchuk, T. Dettmers, M. Ryabinin, Y. Belkada, A. Chumachenko, P. Samygin, and C. Raffel. Petals: Collaborative inference and fine-tuning of large models. *arXiv preprint arXiv:2209.01188*, 2022. URL https://arxiv.org/abs/2209.01188.

A. Clark, D. D. L. Casas, A. Guy, A. Mensch, M. Paganini, J. Hoffmann, B. Damoc, B. Hechtman, T. Cai, S. Borgeaud, G. B. V. D. Driessche, E. Rutherford, T. Hennigan, M. J. Johnson, A. Cassirer, C. Jones, E. Buchatskaya, D. Budden, L. Sifre, S. Osindero, O. Vinyals, M. Ranzato, J. Rae, E. Elsen, K. Kavukcuoglu, and K. Simonyan. Unified scaling laws for routed language models. In *Conference International Conference on Machine Learning*, 2022.

J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. Spanner: Google's globally-distributed database. In *OSDI*, 2012.

S. Dalmia, D. Okhonko, M. Lewis, S. Edunov, S. Watanabe, F. Metze, L. Zettlemoyer, and A. Mohamed. Legonn: Building modular encoder-decoder models, 2023.

J. Dean. Introducing pathways: A next-generation ai architecture. https://blog.google/technology/ai/introducing-pathways-next-generation-ai-architecture/, 2021. Google blog post.

J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, M. a. Ranzato, A. Senior, P. Tucker, K. Yang, Q. Le, and A. Ng. Large scale distributed deep networks. In F. Pereira, C. Burges, L. Bottou, and K. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 25. Curran Associates, Inc., 2012. URL https://proceedings.neurips.cc/paper_files/paper/2012/file/6aca97005c68f1206823815f66102863-Paper.pdf.

A. P. Dempster, N. M. Laird, and D. B. Rubin. Maximum likelihood from incomplete data via the em algorithm. *Journal of the royal statistical society. Series B (methodological)*, pages 1–38, 1977.

A. Douillard, Q. Feng, A. A. Rusu, R. Chhaparia, Y. Donchev, A. Kuncoro, M. Ranzato, A. Szlam, and J. Shen. Diloco: Distributed low-communication training of language models, 2023.

D. Eigen, I. Sutskever, and M. Ranzato. Learning factored representations in a deep mixture of experts. In *Workshop at the International Conference on Learning Representations*, 2014.

W. Fedus, B. Zoph, and N. Shazeer. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity. *CoRR*, abs/2101.03961, 2021. URL https://arxiv.org/abs/2101.03961.

S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. In *SOSP*, 2003.

Google. Effingo: the internal google copy service moving data at scalen.

https://cloud.google.com/blog/products/storage-data-transfer/inside-googles-internal-effingo-data-copy-service/, 2023. Accessed: 2014-02-08.

P. Goyal, P. Dollár, R. Girshick, P. Noordhuis, L. Wesolowski, A. Kyrola, A. Tulloch, Y. Jia, and K. He. Accurate, large minibatch sgd: Training imagenet in 1 hour. *arXiv preprint arXiv:1706.02677*, 2017.

S. Gross, M. Ranzato, and A. Szlam. Hard mixtures of experts for large scale weakly supervised vision. In *CVPR*, 2017.

S. Gururangan, M. Li, M. Lewis, W. Shi, T. Althoff, N. A. Smith, and L. Zettlemoyer. Scaling expert language models with unsupervised domain discovery. *arXiv preprint arXiv:2303.14177*, 2023.

R. A. Jacobs, M. I. Jordan, S. Nowlan, and G. E. Hinton. Adaptive mixtures of local experts. *Neural Computation*, 3:1—-12, 1991.

M. I. Jordan and R. A. Jacobs. Hierarchical mixtures of experts and the em algorithm. *Neural Computation*, 6:181–214, 1994.

D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. *Proceedings of the 3rd International Conference on Learning Representations (ICLR)*, 2014.

T. Kudo and J. Richardson. Sentencepiece: A simple and language independent subword tokenizer and detokenizer for neural text processing. In *EMNLP*, 2018.

D. Lepikhin, H. Lee, Y. Xu, D. Chen, O. Firat, Y. Huang, M. Krikun, N. Shazeer, and Z. Chen. {GS}hard: Scaling giant models with conditional computation and automatic sharding. In *International Conference on Learning Representations*, 2021. URL https://openreview.net/forum?id=qrwe7XHTmYb.

M. Li, S. Gururangan, T. Dettmers, M. Lewis, T. Althoff, N. A. Smith, and L. Zettlemoyer. Branch-train-merge: Embarrassingly parallel training of expert language models. In *First Workshop on Interpolation Regularizers and Beyond at NeurIPS 2022*, 2022. URL https://openreview.net/forum?id=SQgVgE2Sq4.

B. Liu, R. Chhaparia, A. Douillard, S. Kale, A. A. Rusu, J. Shen, A. Szlam, and M. Ranzato. Asynchronous local-sgd training for language modeling, 2024.

D. Narayanan, A. Phanishayee, K. Shi, X. Chen, and M. Zaharia. Memory-efficient pipeline-parallel DNN training. *CoRR*, abs/2006.09503, 2020. URL https://arxiv.org/abs/2006.09503.

OpenAI, :, J. Achiam, S. Adler, S. Agarwal, L. Ahmad, I. Akkaya, F. L. Aleman, D. Almeida, J. Altenschmidt, S. Altman, S. Anadkat, R. Avila, I. Babuschkin, S. Balaji, V. Balcom, P. Baltescu, H. Bao, M. Bavarian, J. Belgum, I. Bello, J. Berdine, G. Bernadett-Shapiro, C. Berner, L. Bogdonoff, O. Boiko, M. Boyd, A.-L. Brakman, G. Brockman, T. Brooks, M. Brundage, K. Button, T. Cai, R. Campbell, A. Cann, B. Carey, C. Carlson, R. Carmichael, B. Chan, C. Chang, F. Chantzis, D. Chen, S. Chen, R. Chen, J. Chen, M. Chen, B. Chess, C. Cho, C. Chu, H. W. Chung, D. Cummings, J. Currier, Y. Dai, C. Decareaux, T. Degry, N. Deutsch, D. Deville, A. Dhar, D. Dohan, S. Dowling, S. Dunning, A. Ecoffet, A. Eleti, T. Eloundou, D. Farhi, L. Fedus, N. Felix, S. P. Fishman, J. Forte, I. Fulford, L. Gao, E. Georges, C. Gibson, V. Goel, T. Gogineni, G. Goh, R. Gontijo-Lopes, J. Gordon, M. Grafstein, S. Gray, R. Greene, J. Gross, S. S. Gu, Y. Guo, C. Hallacy, J. Han, J. Harris, Y. He, M. Heaton, J. Heidecke, C. Hesse, A. Hickey, W. Hickey, P. Hoeschele, B. Houghton, K. Hsu, S. Hu, X. Hu, J. Huizinga, S. Jain, S. Jain, J. Jang, A. Jiang, R. Jiang, H. Jin, D. Jin, S. Jomoto, B. Jonn, H. Jun, T. Kaftan, Łukasz Kaiser, A. Kamali, I. Kanitscheider, N. S. Keskar, T. Khan, L. Kilpatrick, J. W. Kim, C. Kim, Y. Kim, H. Kirchner, J. Kiros, M. Knight, D. Kokotajlo, Łukasz Kondraciuk, A. Kondrich, A. Konstantinidis, K. Kosic, G. Krueger, V. Kuo, M. Lampe, I. Lan, T. Lee, J. Leike, J. Leung, D. Levy, C. M. Li, R. Lim, M. Lin, S. Lin, M. Litwin, T. Lopez, R. Lowe, P. Lue, A. Makanju, K. Malfacini, S. Manning, T. Markov, Y. Markovski,

B. Martin, K. Mayer, A. Mayne, B. McGrew, S. M. McKinney, C. McLeavey, P. McMillan, J. McNeil, D. Medina, A. Mehta, J. Menick, L. Metz, A. Mishchenko, P. Mishkin, V. Monaco, E. Morikawa, D. Mossing, T. Mu, M. Murati, O. Murk, D. Mély, A. Nair, R. Nakano, R. Nayak, A. Neelakantan, R. Ngo, H. Noh, L. Ouyang, C. O'Keefe, J. Pachocki, A. Paino, J. Palermo, A. Pantuliano, G. Parascandolo, J. Parish, E. Parparita, A. Passos, M. Pavlov, A. Peng, A. Perelman, F. de Avila Belbute Peres, M. Petrov, H. P. de Oliveira Pinto, Michael, Pokorny, M. Pokrass, V. Pong, T. Powell, A. Power, B. Power, E. Proehl, R. Puri, A. Radford, J. Rae, A. Ramesh, C. Raymond, F. Real, K. Rimbach, C. Ross, B. Rotsted, H. Roussez, N. Ryder, M. Saltarelli, T. Sanders, S. Santurkar, G. Sastry, H. Schmidt, D. Schnurr, J. Schulman, D. Selsam, K. Sheppard, T. Sherbakov, J. Shieh, S. Shoker, P. Shyam, S. Sidor, E. Sigler, M. Simens, J. Sitkin, K. Slama, I. Sohl, B. Sokolowsky, Y. Song, N. Staudacher, F. P. Such, N. Summers, I. Sutskever, J. Tang, N. Tezak, M. Thompson, P. Tillet, A. Tootoonchian, E. Tseng, P. Tuggle, N. Turley, J. Tworek, J. F. C. Uribe, A. Vallone, A. Vijayvergiya, C. Voss, C. Wainwright, J. J. Wang, A. Wang, B. Wang, J. Ward, J. Wei, C. Weinmann, A. Welihinda, P. Welinder, J. Weng, L. Weng, M. Wiethoff, D. Willner, C. Winter, S. Wolrich, H. Wong, L. Workman, S. Wu, J. Wu, M. Wu, K. Xiao, T. Xu, S. Yoo, K. Yu, Q. Yuan, W. Zaremba, R. Zellers, C. Zhang, M. Zhang, S. Zhao, T. Zheng, J. Zhuang, W. Zhuk, and B. Zoph. Gpt-4 technical report, 2023.

J. Pfeiffer, I. Vulić, I. Gurevych, and S. Ruder. Madx: An adapter-based framework for multi-task cross-lingual transfer. In *EMNLP*, 2020.

J. Pfeiffer, S. Ruder, I. Vulić, and E. M. Ponti. Modular deep learning, 2023.

S. Purushwalkam, M. Nickel, A. Gupta, and M. Ranzato. Task-driven modular networks for zero-shot compositional learning. In *ICCV*, 2019.

C. Raffel. Building machine learning models like open source software. *Communications of the ACM*, 66(2):38–40, 2023a.

C. Raffel. Building machine learning models like open source software. *Commun. ACM*, 66 (2):38–40, jan 2023b. ISSN 0001-0782. doi: 10.1145/3545111. URL https://doi.org/10.1145/3545111.

C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of Machine Learning Research*, 21(1):5485–5551, 2020.

S. Reddi, Z. Charles, M. Zaheer, Z. Garrett, K. Rush, J. Konečný, S. Kumar, and H. B. McMahan. Adaptive federated optimization, 2021.

M. Reisser, C. Louizos, E. Gavves, and M. Welling. Federated mixture of experts, 2021. URL https://openreview.net/forum?id=YgrdmztE4OY.

M. Ryabinin and A. Gusev. Towards crowdsourced training of large neural networks using decentralized mixture-of-experts. *Advances in Neural Information Processing Systems*, 33:3659–3672, 2020.

N. Shazeer, A. Mirhoseini, K. Maziarz, A. Davis, Q. Le, G. Hinton, and J. Dean. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. In *International Conference on Learning Representations*, 2017. URL https://openreview.net/forum?id=B1ckMDqlg.

I. Sutskever, J. Martens, G. Dahl, and G. Hinton. On the importance of initialization and momentum in deep learning. *International Conference on Machine Learning (ICML)*, 2013.

H. Touvron, L. Martin, K. Stone, P. Albert, A. Almahairi, Y. Babaei, N. Bashlykov, S. Batra, P. Bhargava, S. Bhosale, D. Bikel, L. Blecher, C. C. Ferrer, M. Chen, G. Cucurull, D. Esiobu, J. Fernandes, J. Fu, W. Fu, B. Fuller, C. Gao, V. Goswami, N. Goyal, A. Hartshorn, S. Hosseini, R. Hou, H. Inan, M. Kardas, V. Kerkez, M. Khabsa, I. Kloumann, A. Korenev, P. S. Koura, M.-A. Lachaux, T. Lavril, J. Lee, D. Liskovich, Y. Lu, Y. Mao, X. Martinet, T. Mihaylov, P. Mishra, I. Molybog, Y. Nie, A. Poulton,

J. Reizenstein, R. Rungta, K. Saladi, A. Schelten, R. Silva, E. M. Smith, R. Subramanian, X. E. Tan, B. Tang, R. Taylor, A. Williams, J. X. Kuan, P. Xu, Z. Yan, I. Zarov, Y. Zhang, A. Fan, M. Kambadur, S. Narang, A. Rodriguez, R. Stojnic, S. Edunov, and T. Scialom. Llama 2: Open foundation and fine-tuned chat models, 2023.

Acknowledgements

We would like to thank Owen He, Diego de las Casas, Ross Hemsley, Elena Gribovskaya, Yee Whye Teh, Yutian Chen, Alexandre Galashov, Bogdan Damoc, Shreya Pathak, Amal Rannen-Triki, Alek Andreev, David Budden, Jörg Bornschein for their valuable feedback throughout the development of the project. We would also like to thank Jeff Dean, Satinder Baveja, and Raia Hadsell for their suggestions, feedback and support. Finally, we would like to thank Kitty Stacpoole and Guy Scully for their program management support.

Table 4 | **Hyperparameter for Inner Optimization.**

Hyperparameter	150M model	1B model
# Layers	12	24
Hidden dimension	896	2048
# Heads	16	16
Key/value size	64	128
Weight decay	0.1	0.1
Max learning rate	$4e^{-4}$	$2e^{-4}$
Batch size	512	512
# Warmup steps	1000	1000

Supplementary Materials

7.1. Additional Details

We list in Table 4 the hyperparameters used in the inner optimization. The *150M model* hyperparameters are used for both the dense baselines of that size and for the path optimization of DiPaCo. In all our experiments, we used as outer optimizer Nesterov (Sutskever et al., 2013) with outer learning rate of 0.7 and outer momentum of 0.9, following the same recipe introduced by Douillard et al. (2023).

7.2. Routing Details

We first discuss the details of discriminative coarse routing, as used in training. We then discuss the details of more frequent routing, as optionally used in evaluation.

7.2.1. Document level discriminative routing

For discriminative routing, we assume that we have

- Trained an initial language model that can be used as an initialization for finetuning experts.
- Set up a feature extractor that gets a feature for each document to feed to the router.

We alternate between two stages:

1. Train a gater, perhaps based on the PPL of

current experts on an unused portion of the training set,
2. Train experts based on a sharding from current gater.

In this work, the feature for the router is always the average of the hidden state from the last transformer block from the initial LM over the first 32 tokens of a document; let us denote this as $g(\text{document})$. We also reserve a small subset of documents (from here on called the "router data") for making decisions about router quality and training the router; we use .005 of C4 for router data.

Our initial router is always constructed via k-means; and we train the first set of paths using that router (and sharding by argmin to cluster centroid).

To train the discriminative router, we first compute the summed auto-regressive log-likelihood of each document in the router data if encoded by each path. That is, given n documents of length L and paths $f_1, ..., f_K$ we get an $n \times L \times K$ array of scores S_{ijp} where

$$S_{ijp} = \log p_{f_i}(t_{l,j}|t_{l-1,j}, ..., t_{1,j}),$$

and where $t_{i,j}$ is the jth token in the ith document. The router is always trained using a K class linear logistic classifier with

$$\text{argmax}_p \sum_{j=1}^{L} S_{ijp}$$

as the target and $g(\text{document}_i)$ as the feature.

With large numbers of paths, we saw that the paths that were assigned the fewest documents via $\text{argmax}_p \sum_j S_{ijp}$ were even more underrepresented by the output of the trained logistic regressor, and it would often happen that a path was empty when using the output of the regressor. To remedy this, we trained a bias term to match the target document-to-path distribution.

We find that

- One step of training the discriminative router leads to a significant improvement in validation perplexity, more alternating steps lead to further minor improvements.
- Gains are larger with more paths.

7.2.2. *More frequent routing during evaluation*

As before, we have the array S_{ijp} giving the scores of each token in the router data set of documents. We fix a number L, and build a sequence transduction training set where the inputs are the token sequences $t_{i,j}$ from the reserved portion of the training documents, and the outputs $T = T_{ij}$ are defined by

$$T_{ij} = \text{argmax}_p \sum_{j}^{j'} S_{ijp},$$

where $j' = \min(\text{length of document}, j+L-1)$, and L is a window size. Once we have T, we can train a transformer to transduce T from t. We finetune this router from the same model that we fork to train the paths, using the same hyperparameters as our standard LM training (except no warmup), and it converges after a 2K steps. The learned router can then make a decision at any token.

We found that choosing L to be the same size as the number of tokens before re-routing (as in Table 3) led to the best results, but was only marginally better than choosing L large enough so the window always went to the end of the document. The results in Table 3 use this choice of L (the length of the whole sequence, 1024). We then can use the same learned router for any choice of evaluation-time frequency.

In Table 5 we show the effect of different *sharding methods* on a 8×8 DiPaCo. We observe that sharding makes a significant difference, with the discriminative variant yielding the best generalization, with an absolute gain of 0.7 perplexity points over the generative k-means variant. We found that longer training runs yield even greater gain from the discriminative gating.

7.3. More sophisticated generative routing

We explored Product k-Means as generative router for DiPaCo. When k is large, k-Means can become inefficient, because some clusters are assigned few examples. Moreover, in our use

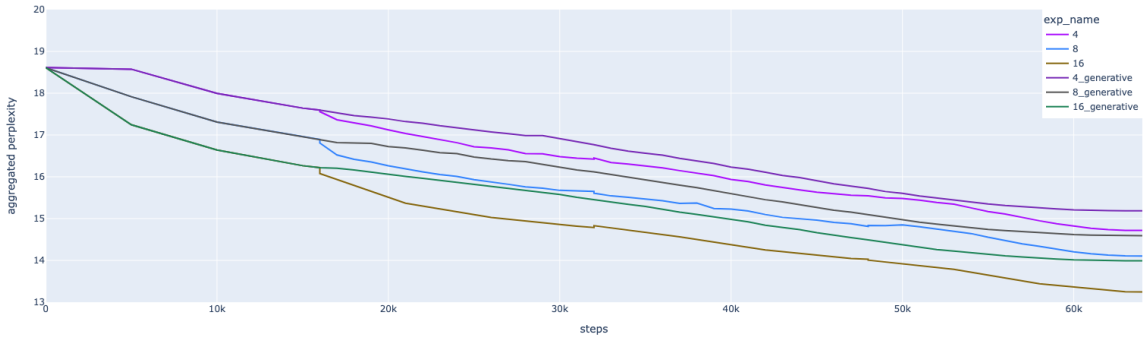

Figure 10 | Generative and discriminative flat MoE models with varying number of paths, three alternating discriminative phases. The "branching" structure of the figure is due to the ancestry of the models: at the far left, all share an initial dense ancestor. At 16K steps, each discriminative-generative pair of experiments shares a generative ancestor, this is continued (restarting the cosine learning rate decay) with the same router ("generative") or with a sequence of new routers ("discriminative")

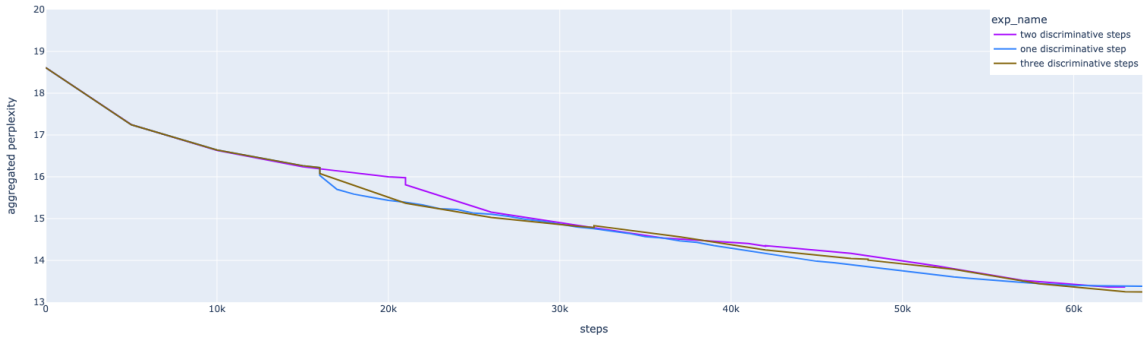

Figure 11 | Validation perplexity vs number of alternating minimization steps, 16 path flat MoE. As the number of alternating steps increases, the results improve, but each step gives less improvement: 14.0 → 13.38 → 13.36 → 13.25 PPL for phases 0, 1, 2, 3 respectively. All discriminative routing results in the main text use one alternating minimization phase.

case, the structure of the shards does not match the structure of the modules. Product k-Means is a way to mitigate these issues. For a 2 level DiPaCo, the feature $z \in \mathcal{R}^D$ is divided in two groups: $z = [z_1, z_2]$, with z_1 comprising the first $D/2$ dimensions, and z_2 the second set of $D/2$ dimensions.[2] We then perform k-means on each set of features independently, and assign each sequence z to the pair (i, j). The first index refers to the assignment of z_1, and the second index to the assignment of z_2 as per equation 1. Although each assignment takes only k possible values, the number of possible unique pair-assignments is k^2. Moreover, the cost of assignment grows with the square root of the number of total pair assignments. We used this approach when experimenting with hundreds of paths; while the results were an improvement over simple k-Means, discriminative routing was better still; and using the product k-means as a router for the initial module training before the discriminative step did not significantly improve results. Note that the discriminative routing adapts to the structure of the DiPaCo, because the paths are used to score documents to find the targets. Nevertheless, we consider alternative generative routing

[2] We could create more groups, but we only used two in this work.

Table 5 | **Sharding Impact on 8x8 DiPaCo**: Validation perplexity after 32 outer optimization steps, each consisting of 62 inner optimization steps with a 8x8 DiPaCo with $P = 64$. Longer training further increases the gap between methods. The discriminative method is based on product k-Means.

Sharding	Validation Perplexity
k-Means	17.2
Product k-Means	16.8
Discriminative	16.5

approaches important for further study, as discriminative scaling by scoring all paths cannot scale to large numbers of paths.