Pipe-Déjàvu: Hardware-aware Latency Predictable, Differentiable Search for Faster Config and Convergence of Distributed ML Pipeline Parallelism

Pengcheng Xu, Kaiyang Chen, Yuanrui Zhang, Gupta Indranil

***Abstract*—Pipe-Déjàvu automates pipeline-parallel training of large deep learning models, emphasizing three main innovations: (1) a predictive model that considers communication cost, model computational cost, and hardware information to predict latency and resources of each parallel configurations. It saves time on pre-profiling before searching the parallel configuration; (2) a differentiable parallel configuration search space inspired by DARTS[39], can potentially reach optimal configuration faster than the original dynamic programming; and (3) parallel random initialization employed for faster training loss convergence. By using hardware-aware scheduling based on latency and resources, Pipe-Déjàvu prioritizes training larger layers on more accessible and better computational devices, shortening the longest training stage and reducing overall training time. Given the time estimation on each stage, Pipe-Déjàvu optimizes the pairing between stage and hardware to ease the longest overhead. With knowledge of hardware capacity and network latency, Pipe-Déjàvu predicts optimal scheduling, saving time and resources from pre-profiling. With differentiable search space, it accelerates searching of parallel configuration. With parallel random initialization, it improves speed of loss convergence.**

I. Introduction

Deep learning has made remarkable advancements in recent years, thanks in part to the increasing computational power of hardware and the development of more efficient algorithms. Despite these advancements, training large-scale neural networks remains a time-consuming and resource-intensive task. To address this challenge, we propose Pipe-Déjàvu, a method that automates pipeline-parallel training of large deep learning models, offering significant improvements in training efficiency and resource utilization.

Pipe-Déjàvu emphasizes three main innovations: (1) a predictive model that considers communication cost, model computational cost, and hardware information to predict latency and resources of each parallel configuration, saving time on pre-profiling before searching the parallel configuration; (2) a differentiable parallel configuration search space inspired by DARTS, which can potentially reach optimal configurations faster than the original dynamic programming; and (3) parallel random initialization employed for faster training loss convergence.

By using hardware-aware scheduling based on latency and resources, Pipe-Déjàvu prioritizes training larger layers on more accessible and better computational devices, shortening the longest training stage and reducing overall training time. Given the time estimation on each stage, Pipe-Déjàvu optimizes the pairing between stage and hardware to ease the longest overhead. With knowledge of hardware capacity and network latency, Pipe-Déjàvu predicts optimal scheduling, saving time and resources from pre-profiling.

In addition to the aforementioned innovations, Pipe-Déjàvu leverages a differentiable search space to accelerate the searching of parallel configurations, further reducing the time required for optimization. Moreover, the parallel random initialization approach improves the speed of loss convergence, contributing to the overall efficiency of the training process.

Our contributions in this paper include:

1) A predictive model that considers communication cost, model computational cost, and hardware information to predict latency and resources of each parallel configuration, saving pre-profiling time.
2) A differentiable parallel configuration search space inspired by DARTS, which accelerates the searching of optimal configurations.
3) Parallel random initialization for faster training loss convergence.
4) Hardware-aware scheduling that prioritizes training larger layers on more accessible and better computational devices, reducing overall training time.

The rest of the paper is organized as follows: Section 2 presents a background on gradient descent, data parallelism, model parallelism, and Bayesian optimization. Section 3 describes the proposed parallel random initialization approach in detail, including its integration with the Bayesian optimization framework. Section 4 discusses the algorithmic implementation of the approach, and Section 5 presents experimental results demonstrating its effectiveness, as well as a comparison to other state-of-the-art methods. Finally, Section 6 concludes the paper and discusses future research directions.

II. Related Work

A. *Data parallelism*

1.1.1 Definition: Data parallelism splits the training data among distributed workers, but copies the model to each worker. Every worker calculates the parameter updates on its own data slice and shares them with other workers before updating the weights. This way, all workers have the same model parameters during training.

1.1.2 Related works: Horovod [18] and PyTorchDDP [20] are two popular data-parallel training systems that use all-reduce to synchronize gradients. BytePS [7, 16] combines all-reduce

and parameter servers and leverages heterogeneous resources in data center clusters. AutoDist [26] employs learning-based methods to devise a data-parallel training strategy. ZeRO [17, 23] enhances the memory usage of data parallelism by reducing duplicated tensors. MiCS [27] lowers the communication scale on top of ZeRO for better scalability on the public cloud.

B. Model/Operator parallelism

1.2.1 Definition: When the model cannot fit in one device, operator parallelism is a viable model parallelism option. Operator parallelism splits the computation of a specific operator, such as "matmul" shown in Appendix Fig. 2b, along non-batch axes, and perform each part of the operator in parallel across multiple devices. Because input tensors are jointly split, when a device performs its op part, the needed portions of input tensors may not be in its local memory. Communication is thus needed to get the input data from other devices. When the tensors are split evenly, i.e., SPMD [24], all devices will follow the same collective communication patterns such as all-reduce, all-gather, and all-to-all.

1.2.2 Related works: The main types of model parallelisms have been discussed in definition. Mesh TensorFlow [19], GSPMD [9,24] and OneFlow [25] provide annotation APIs for users to manually specify the intra-op parallel plan. ColocRL [12] places separate model parts on different devices without pipelining, so the concurrency only happens when there are parallel branches in the model.

C. Pipeline parallelism

1.3.1 Definition: Pipeline parallelism puts different sets of ops from the model graph, called stages, on different workers; at the same time, it divides the training batch into several microbatches, and pipelines the forward and backward passes across microbatches on distributed workers, as Appendix Fig. 2d illustrates. Unlike operator parallelism, pipeline parallelism sends intermediate activations at the forward and backward passes between different workers using point-to-point communication.

1.3.2 Related works: Gpipe [5] divides the input data into micro-batches and forms pipeline parallelisms. PipeDream [13, 14] improves GPipe by using asynchronous training algorithms, reducing memory usage, and integrating it with data parallelism. However, PipeDream is asynchronous while Alpa[28] is a synchronous training system. TeraPipe [11] finds a new pipeline parallelism dimension for transformer-based LMs. Google's Pathway system [2] is a concurrent work of Alpa[28]. Pathway advocates a single controller runtime architecture combining "single program multiple data" (SPMD) and "multiple program multiple data" (MPMD) model. This is similar to Alpa's[28] runtime part, where SPMD is used for intra-op parallelisms and MPMD is used for inter-op parallelism.

D. Manual combination of parallelisms

1.4.1 Definition and related works: The latest developments indicate that the methods discussed above have to be combined to scale up current large DL models [15, 24]. The state-of-the-art training systems, such as Megatron-LM [15, 20], manually create a specific execution plan that merges these parallelisms for transformer language models, which is also called 3D Parallelism. It assumes that the model has the same transformer layer repeated and assigns the same number of layers to each pipeline stage and applies a hand-crafted operator and data parallelism setup uniformly for all layers. It's not generalizable to all models.

E. Automatic combination of parallelisms

1.5.1 Definition: The individual parallelism settings, how they depend on each other and on model and cluster setups, create a complex space that makes it hard to automatically combine these parallelisms. For example, when operator parallelism is used with data parallelism, adding a dataparallel replica means allocating a new set of devices (not just one device) as the worker and finding out the best operator parallelism settings within those devices. When pipeline parallelism is included, the optimal pipelining scheme depends on the data and operator parallelism choices of each pipeline stage and how devices are allocated for each stage. With this view, previous explorations [4, 6, 22, 26] of auto-parallelization are limited to combining data parallelism with at most one model parallelism approach, which misses significant performance opportunities.

1.5.2 Related works: Some research works on how to automatically find the best way to train models in parallel. Tofu [22] uses a dynamic programming method to find the best way to split an operation within a node for linear graphs. FlexFlow [6] defines a "SOAP" problem and uses an MCMC-based random search method. But it only works for placing devices and not for pipeline parallelism. It also cannot handle large graphs or clusters well and does not guarantee optimal. TensorOpt [3] uses a dynamic programming method to find the best way to split an operation that considers both memory and computation cost. Varuna [1] works for clusters with low bandwidth and focuses on finding the best way to use pipeline and data parallelism together. Piper [21] also finds a parallel way that uses both inter- and intra-op parallelism, but it depends on manually designed ways to split operations and assumes a uniform network topology and asynchronous pipeline schedules.

III. Motivation and Concrete Problem Statement

Parallel training has become essential for training today's large models. Parallel training involves distributing the model training process across multiple processing units, such as GPUs or TPUs, which work together to process the data simultaneously. This results in faster convergence and reduces the overall training time. There are many well-studied parallel mechanisms as we describe in above related work, in which pipeline parallel plays an important role in. But pipeline parallel has some inherent shortcomings in fully utilizing all computing units due to sequential dependency of different stages. We can provide a simple example in Fig 2. Assume we divide our

training graph into N stage, stage i generally took t_i to finish, the overall onepass latency for the B batches pipeline is

$$T = \sum_{i=1}^{N} t_i + (B-1) \cdot \max_{1 \le j \le N} \{t_j\}$$

From the above equation we can know that in order to shorten

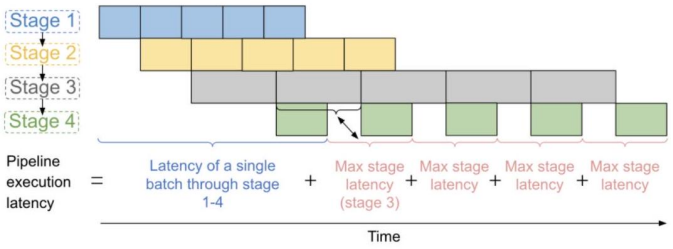


Fig. 1. Example for pipeline parallelism timeline with multiple stages

the overall execution time, our concrete problem is improving the performance (reduce execution time $\max\{t_i\}$) of the longest stage. Instinctively, those longest execution stages are more computational intensive regarding physical or hardware resources like GPU throughput/FLOPS, GPU memory, network latency and even CPU and NVMe capability depending on the user-defined optimization strategy. For example, the sharding strategy introduced by ZeRO incurs heavy communication costs between nodes which emphasize the importance of large network bandwidth. Thus, if we can have some predictor algorithm built in the scheduler, that using information like layer implementation (parameters size / structure) within each stage, hardware conditions of each processing units and user optimization strategy, to map the relatively computational intensive stages to those relatively more "powerful" machine, then we can shorten the execution time of the longest stage compared to random scheduler.

IV. Methods

A. Hard-ware Latency Predict Model for Saving Pre-profiling Time

Fit a function which inputs the (1.Communication cost 2. Model Computation Cost 3. Hardware information), outputs the (1. Estimated execution time 2. Estimated GPU resources to consume), so that we can plan pipeline parallelism without or with less profiling which takes certain part of automatic parallel configuration time.

$$f(communication, model, hardware) = (time, resources)$$

To be more specific, intuitively here is a naive example of how we can model this predicted execution latency for specific ML task mathematically:

$$\begin{aligned} pred\_time(communication, model, hardware) = \\ \alpha \times \frac{g(\#matrix\_multiplication\_operations)}{h(hardware\_computational\_speed)} + \\ \beta \times \frac{p(network\_bandwidth)}{q(data\_to\_transmit)} + C \end{aligned} \tag{1}$$

For the model above, we can learn the unknown parameters or function using some linear or nonlinear regression methods.

B. Differentiable Search Space for Faster Parallel Configuration Search

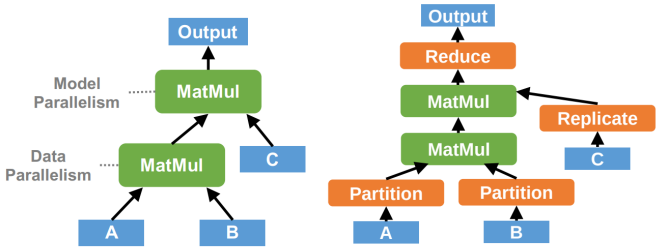
*1) **Dynamic Programming**:* Many existing automatic pipeline parellel use dynamic programming to minimize the training latency given the time estimated for each stage.

$$T^* = \min_{\substack{s_1,\ldots,S_S; \\ (n_1,m_1),\ldots,(n_S,m_S)}} \left\{ \sum_{i=1}^{S} t_i + (B-1) \cdot \max_{1 \le j \le S} \{t_j\} \right\} \tag{2}$$

Alpa's[28] DP algorithm computes the slicing in $O\left(K^3 NM(N + \log(M))\right)$ time for a fixed $t_{\max}$.$t_{\max}$ has at most $O\left(K^2(N + \log(M))\right)$ choices: $t_{\text{intra}}\left((o_i, \ldots, o_j), \text{Mesh}\left(n_s, m_s\right)\right)$ for $i, j = 1, \ldots, K$ and all the submesh choices. The complexity of this DP algorithm is thus $O\left(K^5 NM(N + \log(M))^2\right)$. This complexity is not feasible for a large computational graph of more than ten thousand operators. Therefore, we rethink the search space and make it differentiable to optimize it.

*2) **Parallel Computational Graph Representation**:* To represent the parallel search space as a computational graph, we can model it using a directed acyclic graph (DAG) where each vertex represents a stage of computation and the edges denote the flow of data or model parameters between the stages. The nodes in this graph can be partitioned into different parallelization strategies, such as data parallelism, model parallelism, and pipeline parallelism.

For example, let's consider a simple graph $G(V, E)$, where V is the set of vertices, and E is the set of edges. We can represent a parallel search space for a two-layer neural network as follows:



(a) Computation graph. (b) Parallel computation graph.

Fig. 2. comparing Computation graph and Parallel Computation graph

Both graphs in the figure above describe the same parallelization of two consecutive matrix multiplications $(A \times B) \times C$ (a simplified form of attention). The green and orange boxes denote regular DNN operators and parallelization operators respectively.

- **Data Parallelism (DP)**: In this strategy, each layer of the neural network is replicated across all available devices. The input data is then divided into equal partitions and processed concurrently. The parallel search space for Data Parallelism can be depicted as two distinct sets of nodes, where each set contains nodes corresponding to each device.
- **Model Parallelism (MP)**: In Model Parallelism, each layer of the network is assigned to a different device, distributing the model's layers across multiple devices. This strategy is particularly useful when dealing with large models that exceed the memory capacity of a single device. The parallel search space for this approach can be represented as a single set of nodes, where each node corresponds to a specific layer-device assignment.
- **Pipeline Parallelism (PP)**: Pipeline Parallelism, on the other hand, focuses on improving the computational efficiency of model training by exploiting the pipelined execution of layers across different devices. While it also assigns each layer to a different device, similar to Model Parallelism, the key difference lies in the data processing approach. In Pipeline Parallelism, input data is processed in a pipelined manner across the devices, allowing for concurrent execution of different layers, which reduces the overall training time. The parallel search space for Pipeline Parallelism can be depicted as a sequence of stages, where each stage contains a node representing the layer-device assignment and its position in the pipeline.

To represent both parallelization strategies in the same graph, we can create a graph-like structure with multiple layers and vertices. We can then use the softmax function to assign probabilities to the edges between nodes and layers, which represent the likelihood of selecting a specific parallelization strategy. This continuous representation enables us to differentiate the parallelization search space and optimize the parallel configuration along with the model parameters.

*3) **Differentiable Parallelization Search Space**:* To use a computational graph representation for the differentiable parallelization search space, let's first define a graph $G(V, E)$, where V is the set of vertices and E is the set of edges. In this graph, each vertex represents a computation stage, and the edges denote the flow of data or model parameters between the stages.

For each vertex v_i in V, we associate a set of N parallelization strategies, represented as $\alpha_{i1}, \alpha_{i2}, ..., \alpha_{iN}$. We can then represent the search space as a matrix A of size $|V| \times N$, where A_{ij} denotes the discrete choice for strategy j at stage i. We can transform this search space into a differentiable problem

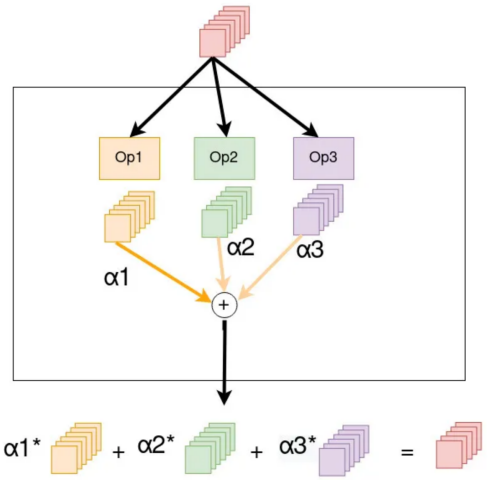


Fig. 3. Softmax Intuition: relaxing the discrete set of candidate operations

by applying the softmax function to each row of A:

$$\text{softmax}(A)_{ij} = \frac{\exp(A_{ij})}{\sum_{k=1}^{N} \exp(A_{ik})} \tag{3}$$

Here, the continuous approximation of the discrete choice is given by $\text{softmax}(A)_{ij}$ for each strategy j at stage i.

Now, we can use this continuous representation of the search space in the context of a computational graph to describe the algebra transformations and parallel strategies. Each vertex v_i in the graph can be associated with a continuous approximation of its parallelization strategy, given by the row $\text{softmax}(A)_i$. These continuous approximations can then be used to compute the forward and backward passes in the computational graph while considering the different parallel strategies.

The rest of the optimization process, including gradient computation and end-to-end optimization, remains the same as in the previous response, with the parallel configuration matrix A being updated during the optimization.

This approach allows us to incorporate a graph representation of the parallelization search space into the differentiable optimization process, enabling the joint optimization of model training and parallelization strategies.

*4) **Bilevel Optimization**:* In the DARTS[39] paper, the authors propose a bilevel optimization problem where they optimize the model's architecture and its weights simultaneously. To adapt this approach for parallelization strategies, we'll introduce a loss function that jointly evaluates the parallel strategy and the training loss.

Let's denote the model's weights by θ and the parallelization strategy represented as a matrix A. We'll define the loss function as $L(\theta, A)$, which is a combination of the training loss and the evaluation of the parallel strategy. We can write the joint optimization problem as:

$$minimize\ L(\theta, A), \quad with\ respect\ to\ \theta, A \tag{4}$$

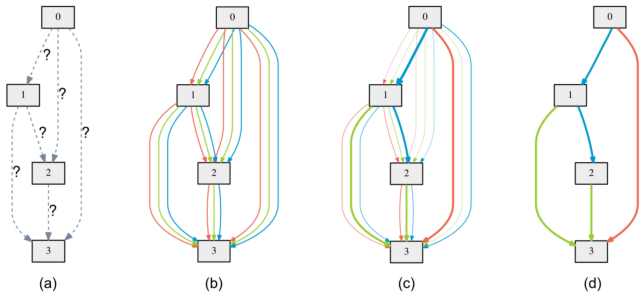


Fig. 4. An overview of Differentiable Search Space: (a) Operations on the edges are initially unknown. (b) Continuous relaxation of the search space by placing a mixture of candidate operations on each edge. (c) Joint optimization of the mixing probabilities and the network weights by solving a bilevel optimization problem. (d) Inducing the final architecture from the learned mixing probabilities.

This bilevel optimization problem can be approximated using gradient-based optimization. The gradients required for updating θ and A are as follows:

$$\begin{aligned} \nabla_\theta L(\theta, A) &= \frac{dL(\theta, A)}{d\theta} \\ \nabla_A L(\theta, A) &= \frac{dL(\theta, A)}{dA} \end{aligned} \tag{5}$$

Here, $\nabla_\theta L(\theta, A)$ is the gradient with respect to the model parameters θ, while $\nabla_A L(\theta, A)$ is the gradient with respect to the parallelization strategy matrix A.

To perform end-to-end optimization, we'll update both the model parameters and the parallelization strategy using gradient descent:

$$\begin{aligned} \theta &= \theta - \eta_\theta \nabla_\theta L(\theta, A) \\ A &= A - \eta_A \nabla_A L(\theta, A) \end{aligned} \tag{6}$$

where η_θ and η_A are the learning rates for the model parameters and the parallelization strategy matrix A, respectively.

This approach allows us to jointly optimize the model training and the parallelization strategies using a gradient-based optimization method similar to the one used in the DARTS[39] paper. Note that in practice, the loss function $L(\theta, A)$ should be designed to effectively balance the trade-offs between training loss and the evaluation of the parallel strategy.

*5) **Complexity Analysis**:* Analyzing the complexity of the differentiable search space inspired by DARTS[39] requires considering the computational costs involved in the forward and backward passes, as well as the optimization process.

Let's break down the complexity analysis:

- Forward pass complexity: The forward pass involves computing the softmax for each vertex in the graph. Since there are $|V|$ vertices, each with N parallelization strategies, the complexity of the forward pass is $O(|V|N)$.
- Backward pass complexity: The gradient computation involves computing the gradients for both the model parameters (θ) and the parallelization strategy matrix (A). Since there are $|V|$ vertices, each with N parallelization strategies, the complexity of computing the gradients with respect to A is $O(|V|N)$. The complexity of computing the gradients with respect to the model parameters θ depends on the specific model and its architecture, but let's denote it by $O(C_\theta)$.
- Optimization complexity: Updating both the model parameters and the parallelization strategy matrix involves applying gradient descent, which has a complexity of $O(|V|N)$ for the parallelization strategy matrix A and $O(C_\theta)$ for the model parameters θ.

In total, the complexity of the differentiable search space inspired by DARTS[39] is:

$$O(|V|N) + O(C_\theta) + O(|V|N) + O(C_\theta) = O(2|V|N + 2C_\theta)$$

However, this complexity analysis doesn't account for the number of iterations required for convergence. If the optimization process takes I iterations, the overall complexity becomes:

$$O(I(2|V|N + 2C_\theta))$$

This complexity analysis shows that the differentiable search space has a linear dependence on the number of vertices, the number of parallelization strategies, and the complexity of the model. It also highlights the importance of the number of optimization iterations for determining the overall complexity of the approach.

C. *Parallel Random Initialization for Faster Loss Convergence*

In addition to the commonly known methods such as data parallelism and model parallelism, we propose a novel approach to accelerate the convergence of the training loss during the optimization of neural networks. This approach involves utilizing parallel random initialization, which refers to the initial randomization of the network parameters.

One way to conceptualize gradient descent is to imagine a rugged hill in search of the minimum value, with gradient descent acting as a ball rolling down the steepest path. Mathematically, the goal of optimization is to minimize the objective function $L(\theta)$, where θ represents the parameters of the neural network. The update rule for gradient descent is given by:

$$\theta_{t+1} = \theta_t - \eta_t \nabla_\theta L(\theta_t) \tag{7}$$

where η_t is the learning rate at time t and $\nabla_\theta L(\theta_t)$ is the gradient of the objective function with respect to the parameters at time t. In data parallelism, n workers are assigned to look at n small directions from a randomly initialized starting point. The gradients are then combined and used for gradient descent.

However, during the initial randomization phase, there may be points where the loss is very low, and gradient descent at those points may quickly reach the minimum. Currently, standard neural network training involves randomization only once at the beginning, rather than sampling multiple times.

In the proposed parallel random initialization approach, we can sample multiple randomization points in parallel at the start of the training process. These points can be filtered based on their respective loss values and gradient magnitudes. Only

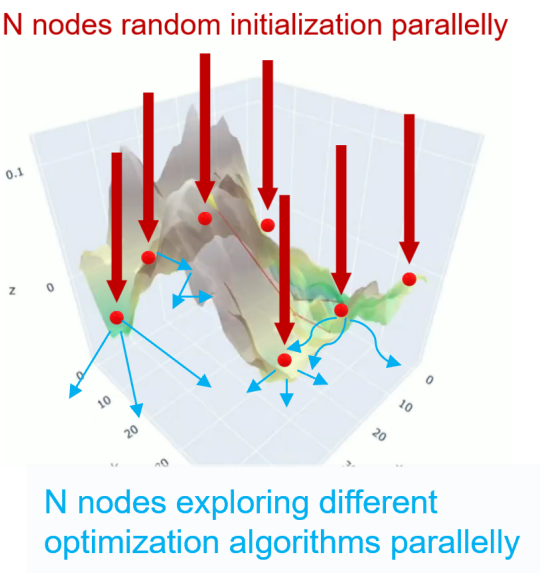


Fig. 5. Parallel Random Initialization and Parallel Optimization Intuition

Algorithm 1 Bayesian Optimization for Parallel Random Initialization

procedure BAYESIANOPTIMIZATION
  function LOSSFUNCTION($params$)
    $initVal \leftarrow$ RESHAPEPARAMS($params, NN()$)
    return PARALLELLOSS($NN(), initVal,$
      $trainLoader, device, epoch, w\_id$)
  end function
  $bounds \leftarrow [(-1, 1)] *$ SUMOFNUMEL($NN()$)
  $result \leftarrow$ GPMINIMIZE($LossFunction,$
    $x0, bounds, numWorkers, randomState$)
  $bestParams \leftarrow result.x$
  return RESHAPEPARAMS($bestParams, NN()$)
end procedure
function RESHAPEPARAMS($params, model$)
  return $\{param.reshape(p.shape) \mid param, p \in zip(params, model.parameters())\}$
end function
function SUMOFNUMEL($model$)
  return $\sum_{p \in model.parameters()} p.numel()$
end function

the ones with small loss and large gradients are retained for further processing.

Subsequently, we can continue to sample within a smaller scope, and further refine the set of points with small loss values. This process can be repeated until a certain level of convergence is achieved, at which point we can focus on the gradient descent of a few selected points.

To implement this approach, we can start with a rough grid with a large interval, and then identify the point with the smallest loss within that grid. We can then perform finer-grained sampling in the vicinity of that point to refine the set of initialization points.

The proposed parallel random initialization approach offers a promising alternative to the standard method of initializing network parameters. By introducing randomness and parallelization during the initialization phase, we can accelerate the convergence of the training loss and potentially reduce the overall training time of neural networks.

D. Model/Pipeline Parallel Fault Tolerance Analysis

Zeno provide a fault-tolerance algorithm from the perspective of SGD for parameter server structure, which is mainly for data parallel. The anaylsis is mainly on the gradient they send. While after literature review, we find that there are very few research about fault tolerance in model parallel and pipeline parallel. Let w_i be the weight assigned to the i^{th} machine, where $\sum_{i=1}^{N} w_i = 1$. The overall gradient is computed as a weighted sum of the local gradients from each machine, where the weights are the weights assigned to each machine:

$$g = \sum_{i=1}^{N} w_i g_i$$

To compute the weight assigned to each machine, we use a suspicion-based approach similar to the one proposed in Zeno. Specifically, we compute the suspicion level of each machine i based on the deviation between its local gradient g_i and the overall gradient g :

$$s_i = \|g_i - g\|$$

The suspicion level s_i represents the degree of deviation between the local gradient g_i and the overall gradient g. Machines with high suspicion levels are likely to be faulty and should be given lower weights during the gradient combination process.

We then update the weight assigned to each machine using the following equation:

$$w_i = \frac{\alpha}{\alpha + s_i}$$

where α is a hyperparameter that controls the weight given to the suspicion level. Higher values of α give greater weight to the suspicion level and vice versa. To identify the k machines with the least influence on the overall gradient, we sort the machines based on their weight in ascending order and select the first k machines. We then replace the gradients of the identified machines with the average of the remaining gradients:

$$\hat{g}_i = \frac{\sum_{j=1}^{N-k} g_j}{N-k}$$

The proposed fault-tolerance algorithm for model parallel/pipeline parallel with alpha-like strategy is designed to mitigate the impact of faulty machines on the overall gradient computation. By identifying and replacing the gradients of the k machines with the least influence on the overall gradient, our approach improves the fault tolerance of model parallel/pipeline parallel and can be used to improve the robustness and reliability of distributed machine learning systems.

V. Experiment Results

A. *Hard-ware Latency Predict model for Saving Pre-profiling Time*

We use one-shot mathematical formula to replace the pre-profiling of alpa[28], having a reduce of profiling time. In the Table below, we demonstrate the pre-profiling time is significant for large model, while our one-shot prediction only took a few minutes.

Models	Time
GPT-39B	$>$ 24hr
Stable Diffusion	$>$ 12hr
GPT2	$>$ 10hr
ViT	$>$ 6hr

Table: Pre-profiling time for different models

The advantage is that it saves profiling time which can be long, up to 30% percent of whole compilation time. The disadvantage is that it can be not so accurate.

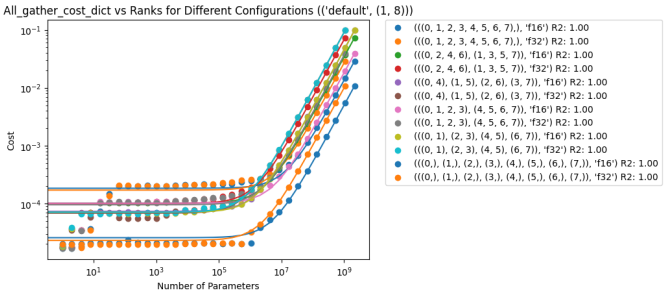


Fig. 6. all gather operation cost measured vs prediction

In figure 6 we have demonstrated the cost for all gather operation. The measured mapping cost under different configuration are represented in dot and our prediction algorithm were drew in solid lines. As number of parameters increase in logarithm, we observed a better fitting line across all configurations. With coefficient of determination reaching 1.0, we proved our regression are capable to predict the cost of placement given appropriate operation and configuration.

B. *Differentiable Search Space for Faster Parallel Configuration Search*

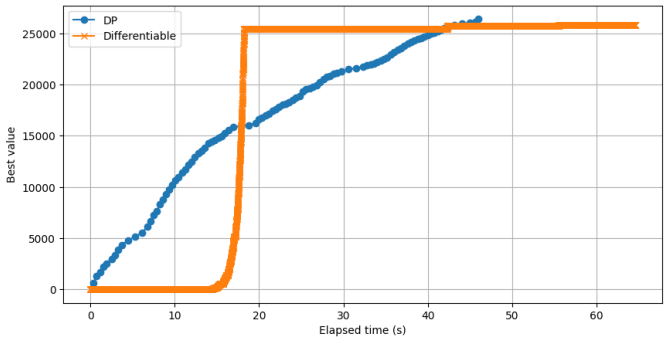


Fig. 7. Differentiable Search results vs Dynamic Programming on Knapsack problem n = 1000,weight_range = (1, 1000),value_range = (1, 100),capacity = 100000. Higher y axis is better.

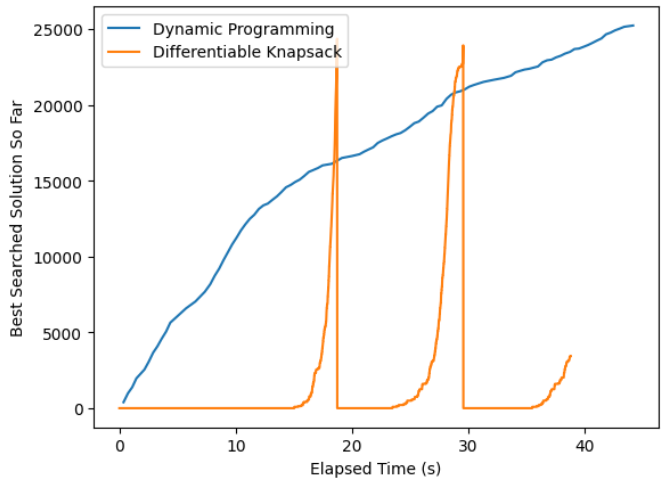


Fig. 8. DP VS Differentiable Search on Knapsack problem n = 1000,weight_range = (1, 1000),value_range = (1, 100),capacity = 100000. Notice that for Differentiable search, this picture plots the current searched solution for specific Time stamp, the accumulated curve is the above one. The DP curve in this plot is still accumulative so far best solution. We can notice the spike of Differentiable Search Algorithm, displaying the pattern of how it searches the optimal.

We first implemented the differentiable search space onto the classic Knapsack problem. We compete it with the classical dynamic programming algorithm. We found that if the search space is quite small, dynamic programming works faster than differentiable method. However, when the search space becomes extremely larger, as the picture shows (when n = 1000,weight range = (1, 1000),value range = (1, 100),capacity = 100000. Higher y axis is better.) , the differentiable dynamic programming finds the suboptimal solution faster than the dynamic programming. It takes only half of the time of dynamic programming to reach the same suboptimal solution. However, due to the gradient descent is not guaranteed to get the global

optimal solution, it seems that our differentiable search stop on suboptimal and it's hard for it to reach global optimal afterwards. It's an interesting phenomenum. It gives us insight that differentiable may be faster at finding suboptimal when the search space is extremely large.

We may later develop a hybrid algorithm, so that it will reach suboptimal fast, then switch to dynamic programming to find the global optimal.

Now we will discuss the dynamic programming of finding parallel placement of distributed ML training. The complexity of Alpa's[28] DP algorithm is $O\left(K^5NM(N+\log(M))^2\right)$. This complexity is not feasible for a large computational graph of more than ten thousand operators.

Therefore, we rethink the search space and use differentiable search space inspired by DARTS[39].

We use differentiable search space instead of dynamic programming, to search optimal parallel configuration faster. We make the complexity become $O\left(I\left(2|V|N+2C_\theta\right)\right)$

I: The number of iterations required for the optimization process to converge.

$|V|$: The number of vertices in the graph representation of the differentiable parallelization search space.

N: The number of parallelization strategies associated with each vertex in the graph.

C_θ: The complexity of computing the gradients with respect to the model parameters θ.

For demonstration purpose, our experiment uses partial code from Alpa's[28] dynamic programming space, to prove that our differentiable search space could potientially outperform the DP method.

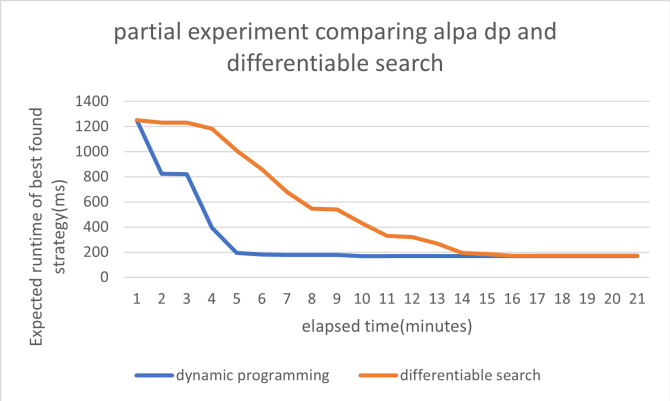


Fig. 9. Differentiable Search results vs Dynamic Programming

The advantage is that it will not like DP, have to cut the search space to make time complexity smaller. It may reach better global optimal. The disadvantage is that the gradient descent may cost more computational power. Its implementation is more complex, can be buggy when implementing the algorithm. Still, whether we will outperform dynamic programming is not clear. We will fully implement the algorithm to see the results in the future. Since the Alpa search space could be large when the number of node and the their computational power is heterogeneous, there could be some cases when the differentiable search method could outperform the dynamic programming method in finding the sub-optimal solution. We will experiment it on more large cluster to see the results.

C. Parallel Random Initialization for Faster Loss Convergence

We use parallel random initialization to select inherently better initialization to make training loss converge faster. We compare different sampling methods like Single Random Initialization(original),Uniform Sampling. We partially implemented Latin hypercube sampling,Bayesian Optimization,Adaptive Sampling, and will test the results of them in the future.

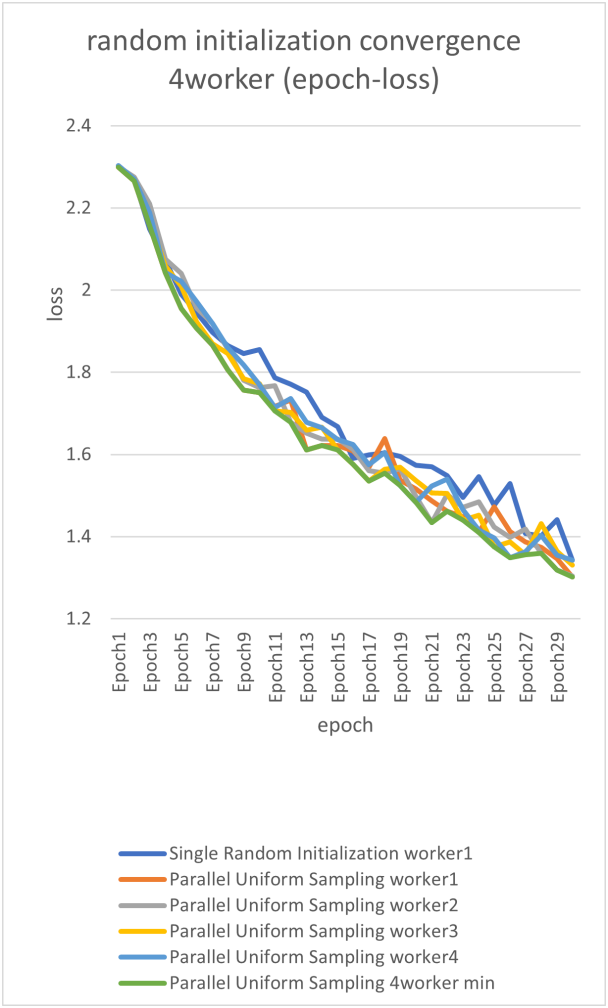


Fig. 10. Parallel Random Initialization Results 4 workers

From the figure and experimental results, when 40workers, the convergence speed can get around 1.5% improve. That's a small improve but it shows it may work if we improve our initialization or sampling method. In the future we will test different init method like kaiming_normal or xavier_normal, and different sampling methods like Latin hypercube sampling,Bayesian Optimization,Adaptive Sampling. The advantage is that if all methods is used up, this method

can drain the resource to gain faster loss convergence. The disadvantage is that it will cost many resources.

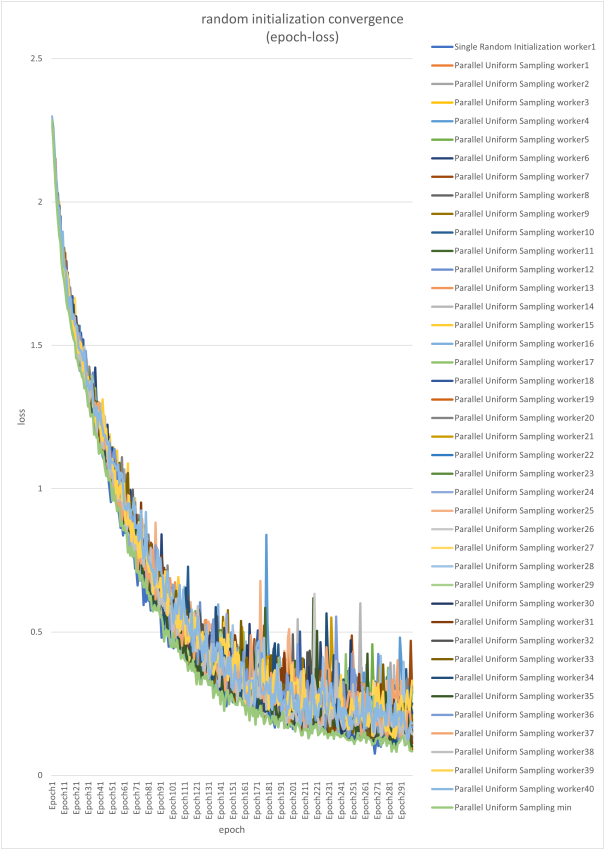


Fig. 12. Parallel Random Initialization Results 40 workers

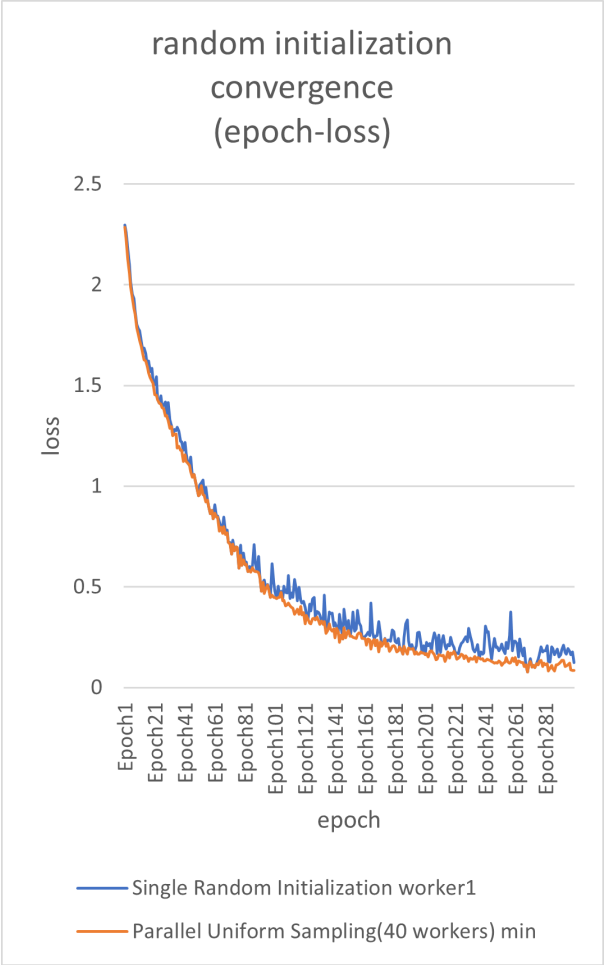


Fig. 13. Parallel Random Initialization Results 40 workers

VI. Timeline/Expected Milestones for achieving our goals

4.1 Mar 15th: Get some initial experiment results that prove our motivations(detect the idle bubble in pipeline flow).

4.2 Mar,30th: Figured out the concrete approaches for predictor & finish most part of mid-term report.

4.3 Apr,15th: Implemented a simple heuristic binding scheduler (maybe only taken several simple factors like network) based on existing codebase like ColossalAI, Alpa[28]. Experiment Differentiable Search on small part of parallel configuration. Experiment Parallel Random Initialization.

4.4 Apr,30th: Implemented the fully functional hard-ware aware binding algorithm between pipeline stages and computation meshes. Experiment Differentiable Search on larger part of parallel configuration. Experiment Parallel Random Initialization on different init strategy and random sampling algorithm.

4.5 May,6th: Perform more training to evaluate our system and finish final report

Appendix A
References

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

[2] Paul Barham, Aakanksha Chowdhery, Jeff Dean, Sanjay Ghemawat, Steven Hand, Daniel Hurt, Michael Isard, Hyeontaek Lim, Ruoming Pang, Sudip Roy, et al. Pathways: Asynchronous distributed dataflow for ml. Proceedings of Machine Learning and Systems, 4, 2022.

[3] Zhenkun Cai, Xiao Yan, Kaihao Ma, Yidi Wu, Yuzhen Huang, James Cheng, Teng Su, and Fan Yu. Tensoropt: Exploring the tradeoffs in distributed dnn training with auto-parallelism. IEEE Transactions on Parallel and Distributed Systems, 33(8):1967-1981, 2021.

[4] Shiqing Fan, Yi Rong, Chen Meng, Zongyan Cao, Siyu Wang, Zhen Zheng, Chuan Wu, Guoping Long, Jun Yang, Lixue Xia, et al. Dapple: A pipelined data parallel approach for training large models. In Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pages 431-445, 2021.

[5] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, et al. Gpipe: Efficient training of giant neural networks using pipeline parallelism. Advances in neural information processing systems, 32:103-112, 2019. [6] Zhihao Jia, Matei Zaharia, and Alex Aiken. Beyond data and model parallelism for deep neural networks. arXiv preprint arXiv:1807.05358, 2018.

[7] Yimin Jiang, Yibo Zhu, Chang Lan, Bairen Yi, Yong Cui, and Chuanxiong Guo. A unified architecture for accelerating distributed dnn training in heterogeneous gpu/cpu clusters. In 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20), pages 463-479, 2020.

[8]Soojeong Kim, Gyeong-In Yu, Hojin Park, Sungwoo Cho, Eunji Jeong, Hyeonmin Ha, Sanha Lee, Joo Seong Jeong, and Byung-Gon Chun. Parallax: Sparsity-aware data parallel training of deep neural networks. In Proceedings of the Fourteenth EuroSys Conference 2019, pages 1-15, 2019. [9] Dmitry Lepikhin, HyoukJoong Lee, Yuanzhong Xu, Dehao Chen, Orhan Firat, Yanping Huang, Maxim Krikun, Noam Shazeer, and Zhifeng Chen. Gshard: Scaling giant models with conditional computation and automatic sharding. arXiv preprint arXiv:2006.16668, 2020

[10] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, et al. Pytorch distributed: Experiences on accelerating data parallel training. arXiv preprint arXiv:2006.15704, 2020.

[11] Zhuohan Li, Siyuan Zhuang, Shiyuan Guo, Danyang Zhuo, Hao Zhang, Dawn Song, and Ion Stoica. Terapipe: Token-level pipeline parallelism for training large-scale language models. arXiv preprint arXiv:2102.07988, 2021.

[12] Azalia Mirhoseini, Hieu Pham, Quoc V Le, Benoit Steiner, Rasmus Larsen, Yuefeng Zhou, Naveen Kumar, Mohammad Norouzi, Samy Bengio, and Jeff Dean. Device placement optimization with reinforcement learning. In International Conference on Machine Learning, pages 2430-2439. PMLR, 2017.

[13] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R Devanur, Gregory R Ganger, Phillip B Gibbons, and Matei Zaharia. Pipedream: generalized pipeline parallelism for dnn training. In Proceedings of the 27th ACM Symposium on Operating Systems Principles, pages $1-15, 2019$.

[14] Deepak Narayanan, Amar Phanishayee, Kaiyu Shi, Xie Chen, and Matei Zaharia. Memoryefficient pipelineparallel dnn training. In International Conference on Machine Learning, pages 79377947. PMLR, 2021.

[15] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, et al. Efficient large-scale language model training on gpu clusters using megatron-lm. In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, pages 1-15, 2021. [16] Yanghua Peng, Yibo Zhu, Yangrui Chen, Yixin Bao, Bairen Yi, Chang Lan, Chuan Wu, and Chuanxiong Guo. A generic communication scheduler for distributed dnn training acceleration. In Proceedings of the 27th ACM Symposium on Operating Systems Principles, pages 16-29, 2019. [17] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. Zero: Memory optimizations toward training trillion parameter models. In SC20: International Conference for High Performance Computing, Networking, Storage and Analysis, pages 1-16. IEEE, 2020. [18]Alexander Sergeev and Mike Del Balso. Horovod: fast and easy distributed deep learning in tensorflow. arXiv preprint arXiv:1802.05799, 2018.

[19] Noam Shazeer, Youlong Cheng, Niki Parmar, Dustin Tran, Ashish Vaswani, Penporn Koanantakool, Peter Hawkins,

HyoukJoong Lee, Mingsheng Hong, Cliff Young, et al. Meshtensorflow: Deep learning for supercomputers. arXiv preprint arXiv:1811.02084, 2018.

[20] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism. arXiv preprint arXiv:1909.08053, 2019.

[21] Jakub M Tarnawski, Deepak Narayanan, and Amar Phanishayee. Piper: Multidimensional planner for dnn parallelization. Advances in Neural Information Processing Systems, 34, 2021. [22] Minjie Wang, Chien-chin Huang, and Jinyang Li. Supporting very large models using automatic dataflow graph partitioning. In Proceedings of the Fourteenth EuroSys Conference 2019, pages 1-17, 2019.

[23] Yuanzhong Xu, HyoukJoong Lee, Dehao Chen, Hongjun Choi, Blake Hechtman, and Shibo Wang. Automatic cross-replica sharding of weight update in data-parallel training. arXiv preprint arXiv:2004.13336, 2020.

[24] Yuanzhong Xu, HyoukJoong Lee, Dehao Chen, Blake Hechtman, Yanping Huang, Rahul Joshi, Maxim Krikun, Dmitry Lepikhin, Andy Ly, Marcello Maggioni, et al. Gspmd: General and scalable parallelization for ml computation graphs. arXiv preprint arXiv:2105.04663, 2021

[25] Jinhui Yuan, Xinqi Li, Cheng Cheng, Juncheng Liu, Ran Guo, Shenghang Cai, Chi Yao, Fei Yang, Xiaodong Yi, Chuan Wu, et al. Oneflow: Redesign the distributed deep learning framework from scratch. arXiv preprint arXiv:2110.15032, 2021.

[26] Hao Zhang, Yuan Li, Zhijie Deng, Xiaodan Liang, Lawrence Carin, and Eric Xing. Autosync: Learning to synchronize for data-parallel distributed deep learning. Advances in Neural Information Processing Systems, 33, 2020.

[27] Zhen Zhang, Shuai Zheng, Yida Wang, Justin Chiu, George Karypis, Trishul Chilimbi, Mu Li, and Xin Jin. Mics: Near-linear scaling for training gigantic model on public cloud. arXiv preprint arXiv:2205.00119, 2022.

[28]Zheng, Lianmin, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang et al. "Alpa: Automating Inter-and Intra-Operator Parallelism for Distributed Deep Learning." In 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22), pp. 559578. 2022.

[29]https://www.deepspeed.ai/tutorials/pipeline/

[30]https://huggingface.co/docs/transformers/v4.15.0/parallelism

[31]https://www.usenix.org/system/files/osdi22-zheng-lianmin.pdf

[32]Rapp, Martin, Ramin Khalili, Kilian Pfeiffer, and Jörg Henkel. "Distreal: Distributed resource-aware learning in heterogeneous systems." In Proceedings of the AAAI Conference on Artificial Intelligence, vol. 36, no. 7, pp. 8062-8071. 2022. https://arxiv.org/abs/2112.08761v1

[34]Amaris, Marcos, Raphael Camargo, Daniel Cordeiro, Alfredo Goldman, and Denis Trystram. "Evaluating execution time predictions on GPU kernels using an analytical model and machine learning techniques." Journal of Parallel and Distributed Computing 171 (2023): 66-78. https://www.sciencedirect.com/science/article/pii/S0743731522001903?casa token=Kqi7243EJsAAAAA:P9NoXtKwokKF11fN8OgGYys5r0gaF gVweA7ZyMfYvSMkUkGQ37oMASLi49cnXp 0F9 Oogk3Fg

[35]Pham, Thanh-Phuong, Juan J. Durillo, and Thomas Fahringer. "Predicting workflow task execution time in the cloud using a two-stage machine learning approach." IEEE Transactions on Cloud Computing 8, no. 1 (2017): 256-268.

[36]Liu, Hanxiao, Karen Simonyan, and Yiming Yang. "Darts: Differentiable architecture search." arXiv preprint arXiv:1806.09055 (2018).

[37]T. -P. Pham, J. J. Durillo and T. Fahringer, "Predicting Workflow Task Execution Time in the Cloud Using A Two-Stage Machine Learning Approach," in IEEE Transactions on Cloud Computing, vol. 8, no. 1, pp. 256-268, 1 Jan.-March 2020, doi: 10.1109/TCC.2017.2732344.

APPENDIX B
CODE

The source code of this project is at https://github.com/explcre/pipeDejavu

APPENDIX C
DISTRIBUTED TRAINING FAULT TOLERANCE ALGORITHM

Algorithm 2 Distributed Training Fault Tolerance Algorithm

Initialize the weights of each machine $w_i = \frac{1}{N}$
while not converged **do**
    Compute local gradients g_i on each machine
    Compute overall gradient $g = \sum_{i=1}^{N} w_i g_i$
    Compute suspicion level of each machine $s_i = |g_i - g|$
    Update the weight of each machine $w_i = \frac{\alpha}{\alpha + s_i}$
    Sort machines based on their weight in ascending order
    Identify the k machines with the least influence on the overall gradient
    Replace gradients of identified machines with the average of remaining gradients $\hat{g}i = \frac{\sum j=1^{N-k} g_j}{N-k}$
    Compute overall gradient $g = \sum_{i=1}^{N} w_i \hat{g}_i$
    Update model parameters using the overall gradient
end while

APPENDIX D
IMAGES OF DIFERENT PARALLEL PARADIGM

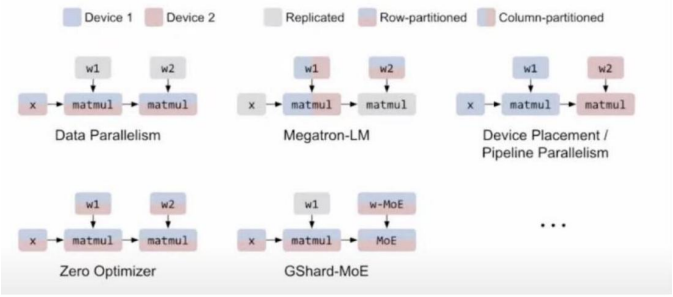


Fig. 14. Existing Parallelization Techniques

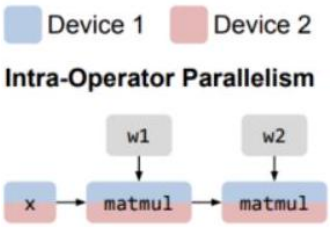


Fig. 15. (a) Data Parallelism

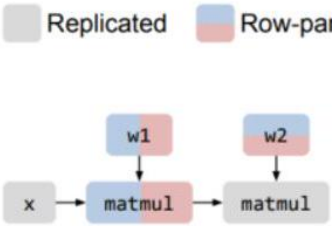


Fig. 16. (b) Operator Parallelism

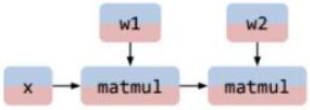


Fig. 17. (c) ZeRO Optimizer

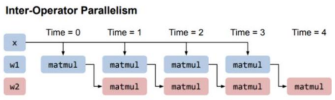


Fig. 18. (d) Pipeline Parallelism

Figure 2: Common parallelization techniques for training a 2-layer Multi-layer Perceptron (MLP). Only the forward pass is shown. " x " is the input data. "w1" and "w2" are two weight matrices.

APPENDIX E
LOGIC FLOW OF HOW TO PARALLELIZE A COMPUTATIONAL GRAPH

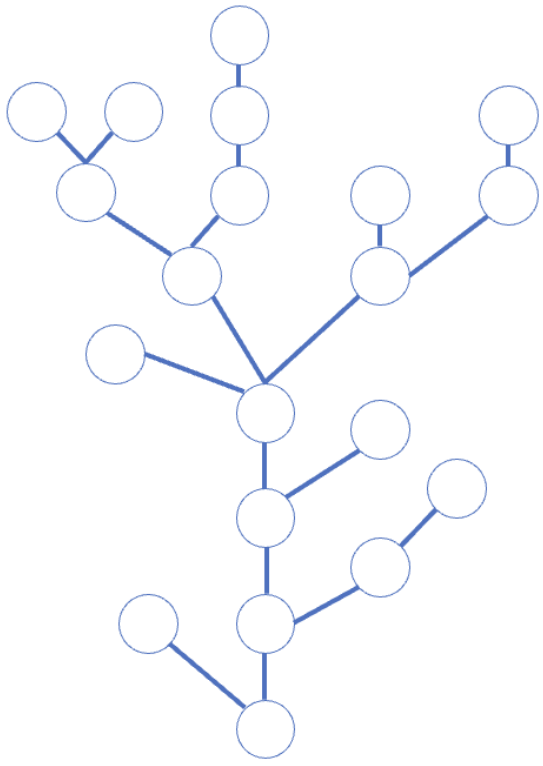


Fig. 18. Step 1: Computation Graph when node number n = 4

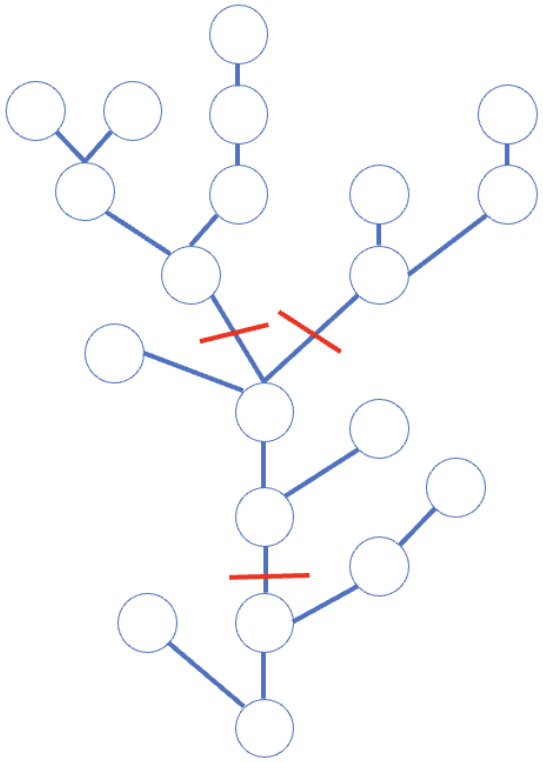


Fig. 19. Step 2: Cut n-1 of this computation graph

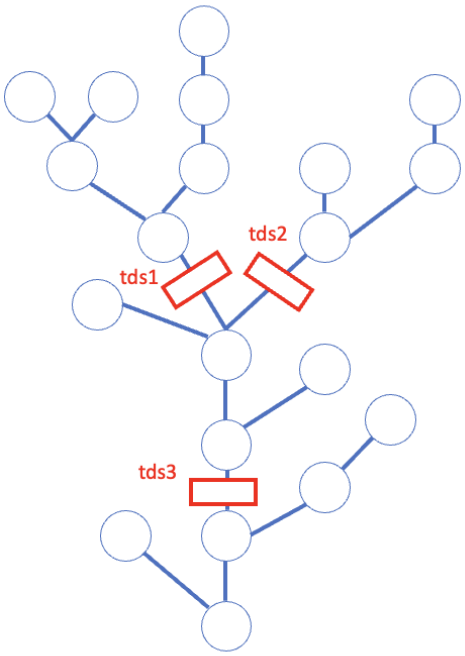


Fig. 20. Step 3: Add communication cost

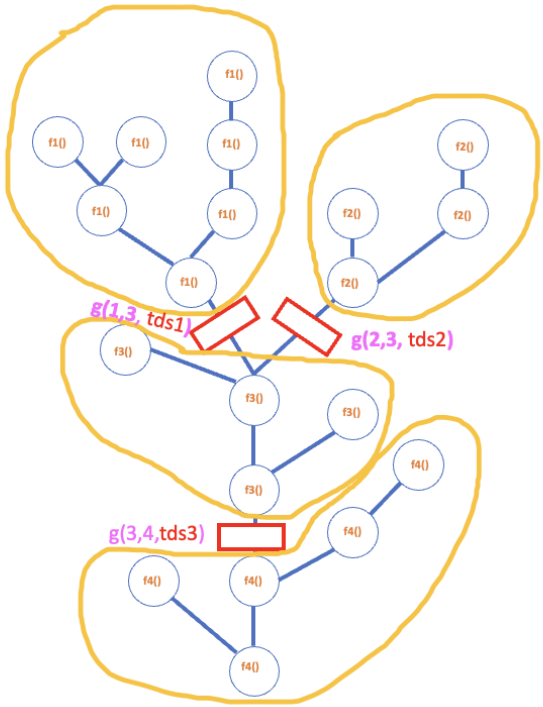


Fig. 21. Step 4: Apply hardware cost function

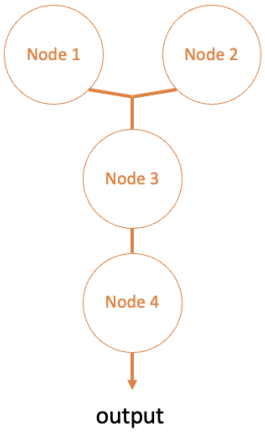


Fig. 22. Step 5: Abstract to n nodes

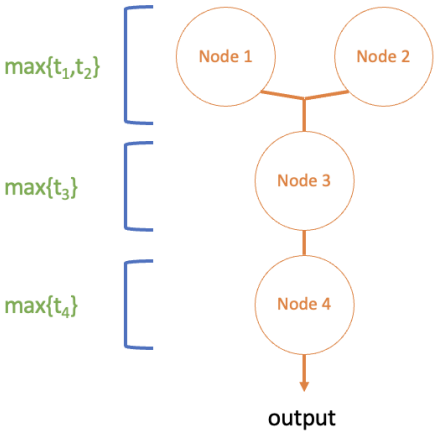


Fig. 23. Step 6: Apply MAX operator to each layer of the tree

APPENDIX F
CODE EXAMPLE IN FITTING PROFILING DATABASE PICKLE FILE USING LINEAR REGRESSION

Listing 1. Code Example in fitting profiling database pickle file using linear regression

```
import pickle
import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression
from sklearn.metrics import r2_score
from decimal import Decimal
from sklearn.preprocessing import StandardScaler
# Load the provided data
with open("prof_database.pkl", "rb") as f:
    data = pickle.load(f)

# Function to extract X and Y values from the dataset
def extract_data(dataset, key):
    x = [point[0] for point in dataset[key]]
    #x = [Decimal(point[0]) for point in dataset[key]]
    y = [point[1] for point in dataset[key]]
    #y = [Decimal(point[1]) for point in dataset[key]]

    # Replace extreme values with the maximum finite representable value for float64
    x = np.clip(x, np.finfo(np.float64).min, np.finfo(np.float64).max)
    y = np.clip(y, np.finfo(np.float64).min, np.finfo(np.float64).max)

     # Replace NaN values with the mean of the non-NaN elements in the array
    y = np.where(np.isnan(y), np.nanmean(y), y)

    return np.array(x).reshape(-1, 1), np.array(y)

# Function to filter out infinity or large values from X and y
def filter_data(X, y):
    X_flat = np.ravel(X)
    overflow_mask = (X_flat < np.finfo(np.float32).max) & (y < np.finfo(np.float32).max)
    return X[overflow_mask], y[overflow_mask]

# Function to apply log transformation to y values
def apply_log_transform(y):
    return np.log(y)

# List of attributes
attributes = [
    'all_gather_cost_dict',
    'all_reduce_cost_dict',
    'all_to_all_cost_dict',
```

```
    'reduce_scatter_cost_dict',
    'available_memory_per_device',
    'dot_cost_dict',
    'conv_cost_dict',
    'op_cost_dict',
]

# Loop through all keys
for key in data.keys():
    # Loop through all attributes
    for attr in attributes:
        attribute_dict = getattr(data[key], attr)

        if not isinstance(attribute_dict, dict):
            continue

        for config, cost_data in attribute_dict.items():
            X, y = extract_data(attribute_dict, config)

            # Filter out infinity or large values from X and y
            X, y = filter_data(X, y)
            # Apply log transformation to y values
            y = np.where(np.isnan(y), np.nanmean(y), y)
            # Initialize the scaler
            scaler = StandardScaler()
            if X.size == 0 or y.size == 0:
                print(f"Empty arrays encountered for {key}, {attr}, {config}. Skipping...")
                continue

            y_norm = scaler.fit_transform(y.reshape(-1, 1)).reshape(-1)
            # Linear Regression
            lr = LinearRegression()
            lr.fit(X, y_norm)
            y_pred_norm = lr.predict(X)

            # Rescale predictions back to original scale
            y_pred = scaler.inverse_transform(y_pred_norm.reshape(-1, 1)).reshape(-1)
            r2 = r2_score(y, y_pred)
            '''
            # Linear Regression
            lr = LinearRegression()
            lr.fit(X, y)
            y_pred = lr.predict(X)
            r2 = r2_score(y, y_pred)
            '''
            # Print accuracy results
            print(f"Key: {key}")
            print(f"Attribute: {attr}")
            print(f"Configuration: {config}")
            print(f"R2 score: {r2:.2f}")
            print(f"Slope: {lr.coef_[0]}")
            print(f"Intercept: {lr.intercept_}\n")

            # Visualization
            plt.scatter(X, y, label=f"{config} R2: {r2:.2f}")
            plt.plot(X, y_pred)

        plt.xscale("log")
        plt.yscale("log")
        plt.xlabel("Number of Parameters")
        plt.ylabel("Cost")

        # Move the legend outside of the plot
        plt.legend(bbox_to_anchor=(1.05, 1), loc='upper left', borderaxespad=0.)

        plt.title(f"{attr.capitalize()} vs Ranks for Different Configurations ({key})")

        # Save the plot
        plt.savefig(f"{attr}_vs_ranks_{key}.png", bbox_inches='tight')

        # Show the plot
        plt.show()

        # Clear the plot for the next attribute
        plt.clf()
```

APPENDIX G
PREDICTIVE MODEL RESULTS FOR PRE-PROFILING STAGE

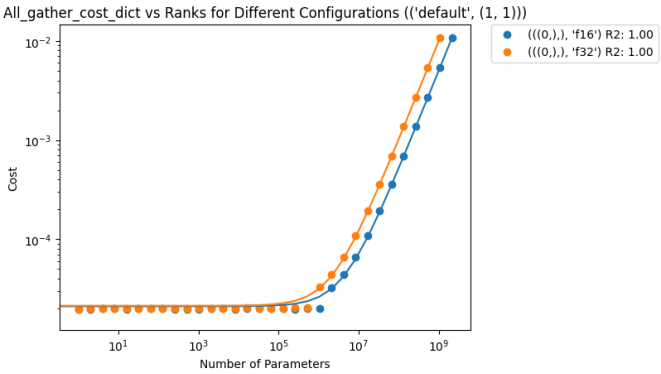


Fig. 24.

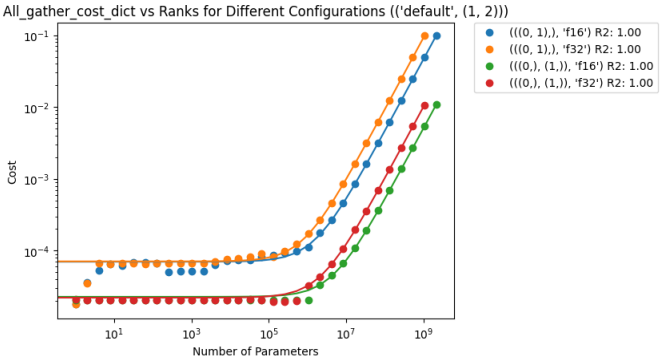


Fig. 25.

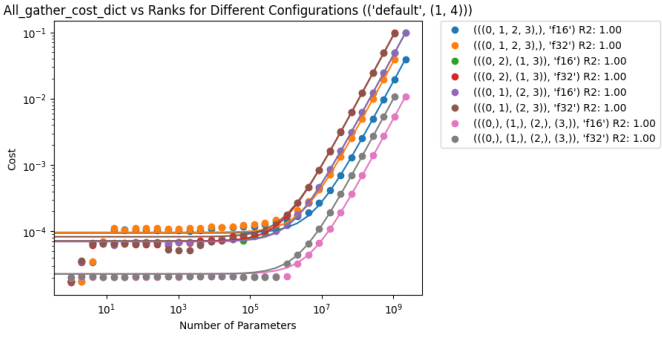


Fig. 26.

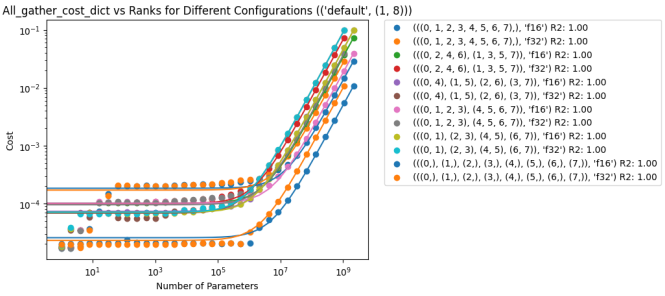


Fig. 27.

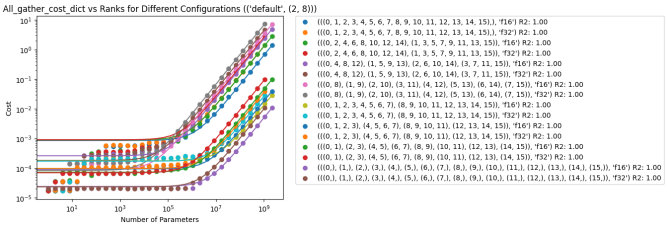


Fig. 28.

Fig. 29.

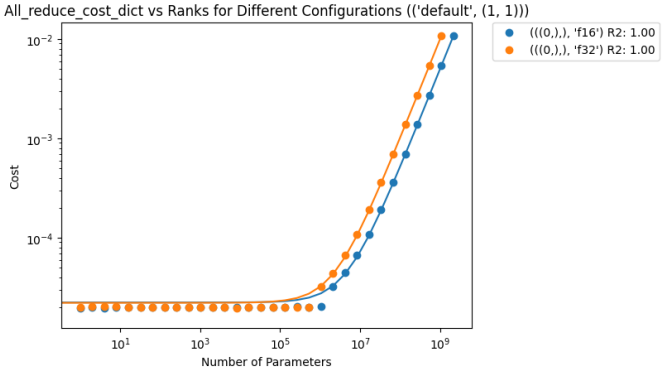


Fig. 30.

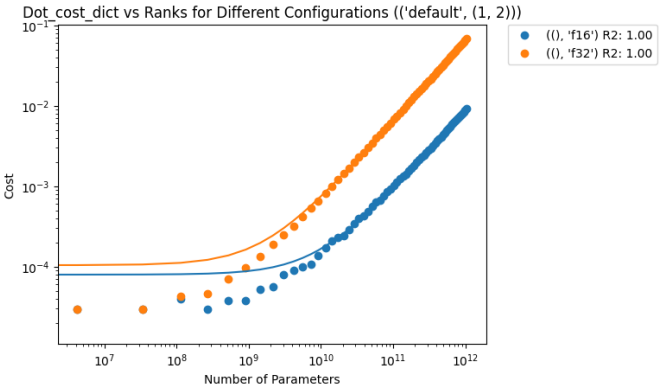


Fig. 31.

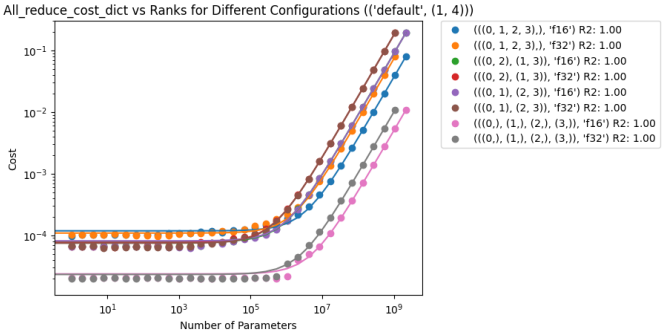


Fig. 32.

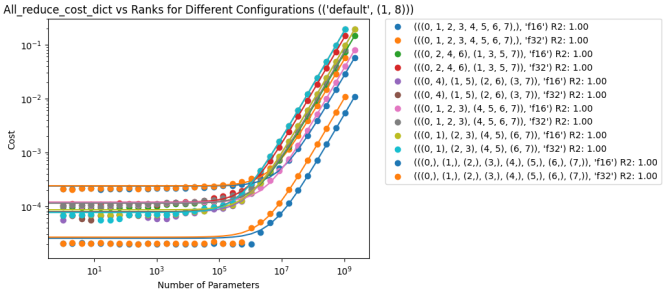


Fig. 33.

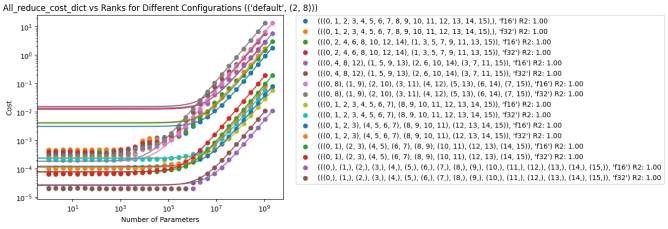


Fig. 34.

Fig. 35.

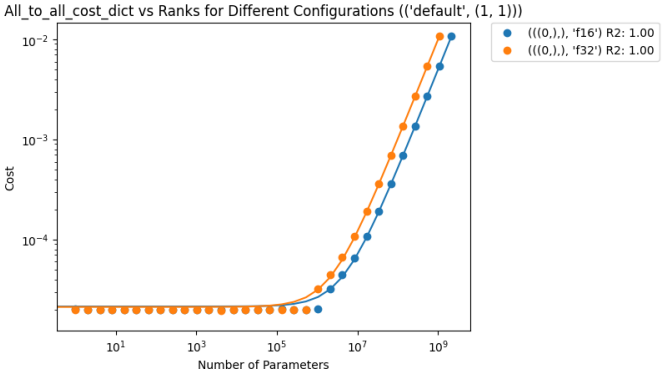


Fig. 36.

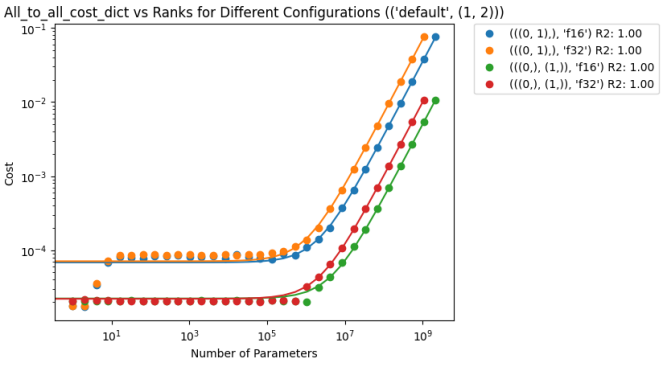


Fig. 37.

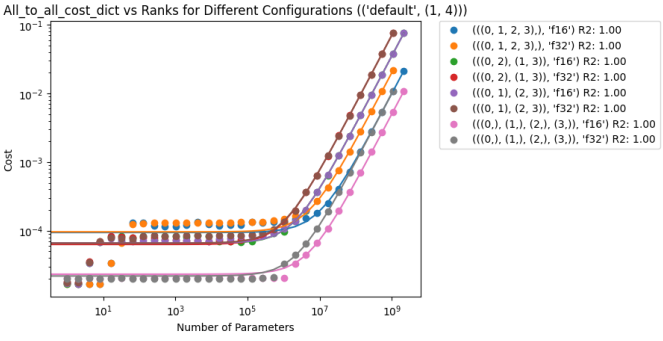


Fig. 38.

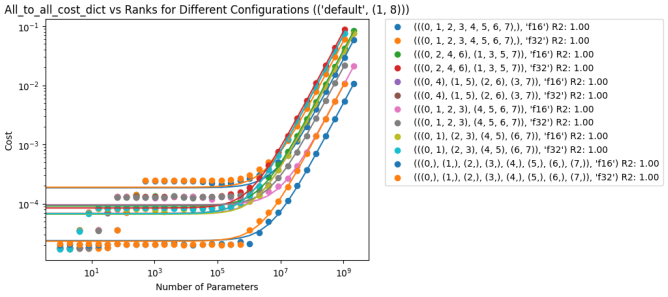


Fig. 39.

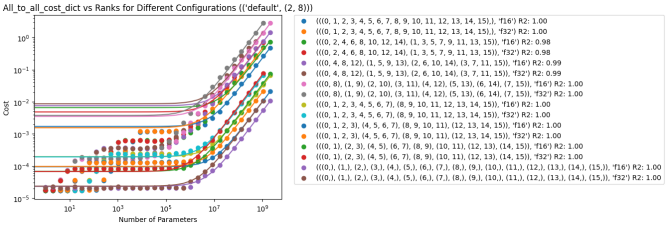


Fig. 40.

Fig. 41.

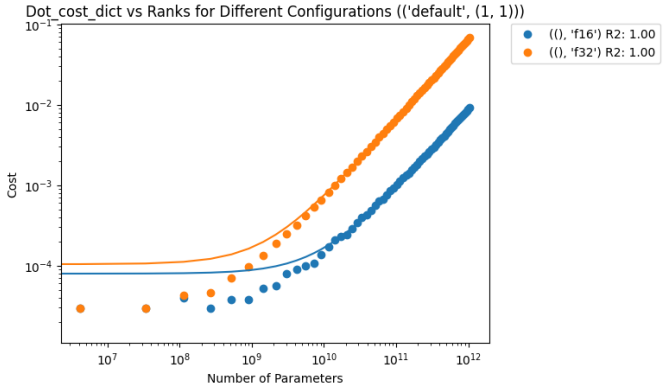


Fig. 42.

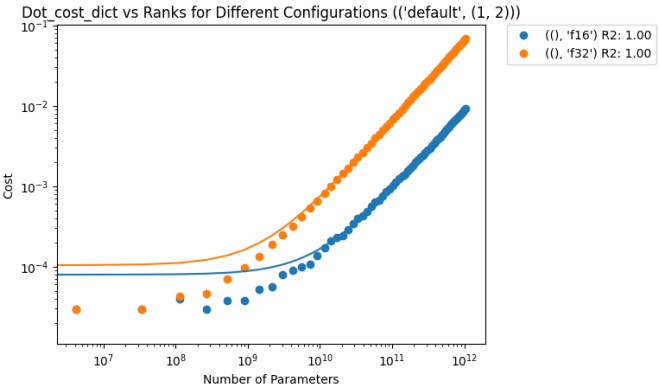


Fig. 43.

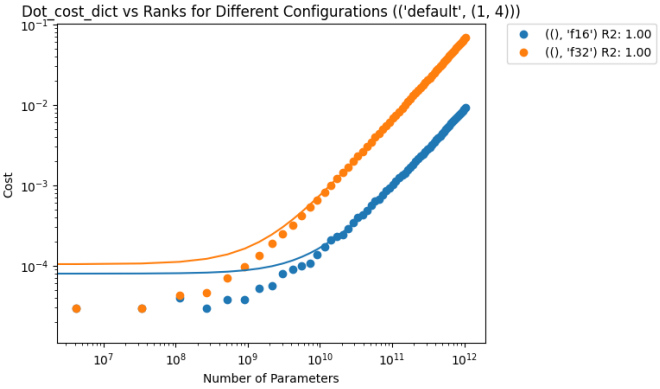


Fig. 44.

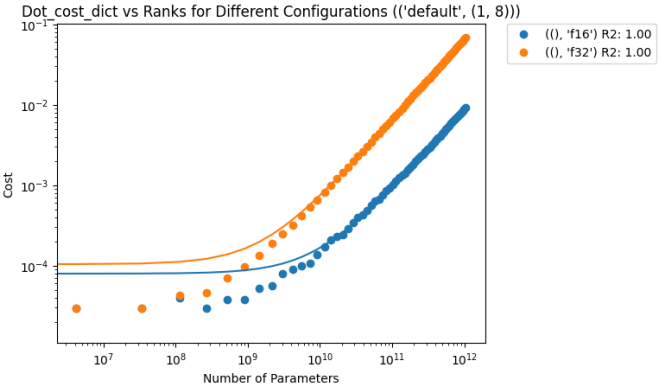


Fig. 45.

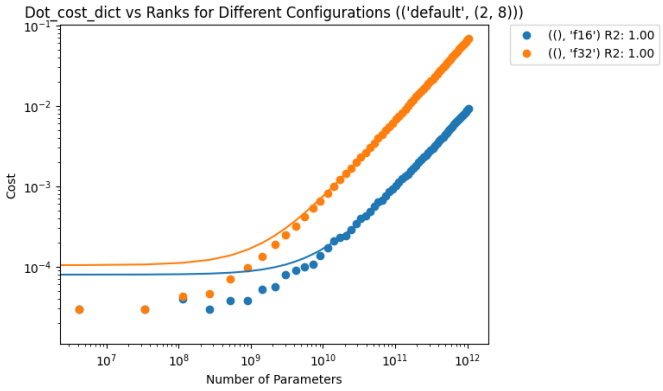


Fig. 46.

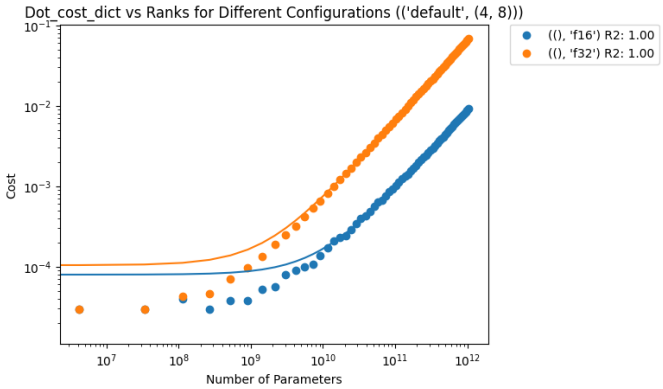


Fig. 47.

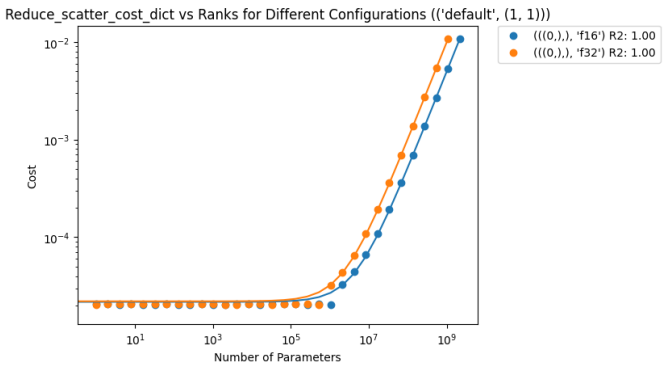


Fig. 48.

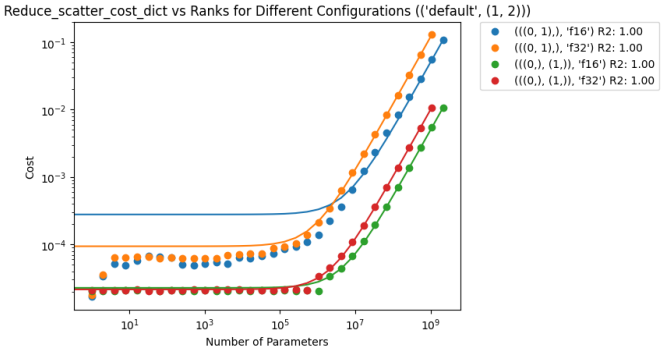


Fig. 49.

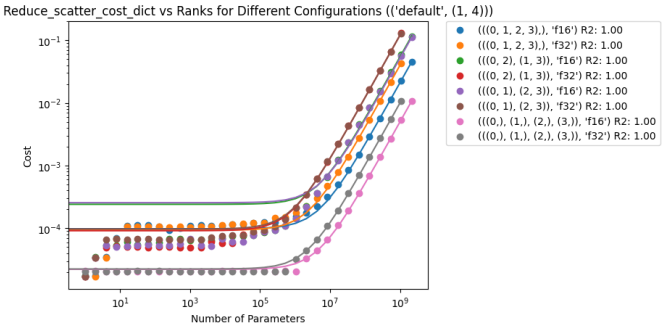


Fig. 50.

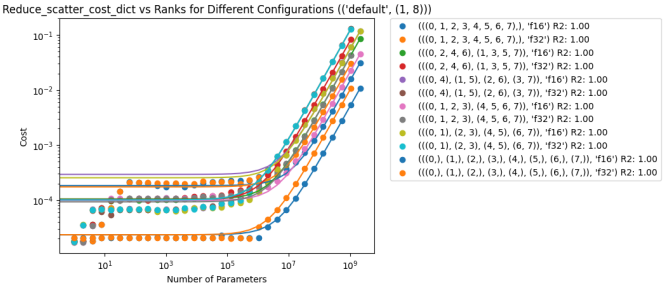


Fig. 51.

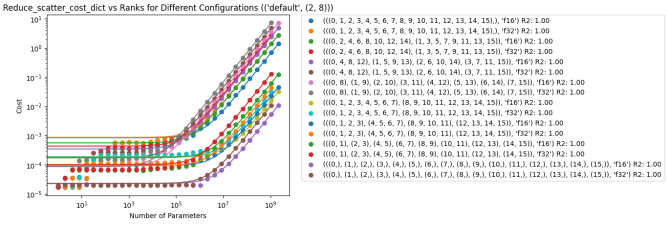


Fig. 52.

Fig. 53.

APPENDIX H
EXAMPLE CODE FOR DP AND DIFFERENTIABLE SEARCH SPACE FOR CLASSIC KNAPSACK PROBLEM

Listing 2. Dynamic Programming Knapsack Algorithm

```python
def knapsack(items, capacity, print_every=500, timer_every=10):
    n = len(items)
    dp = [[0 for _ in range(capacity + 1)] for _ in range(n + 1)]

    best_solutions_dp = []

    start_time = time.time()
    for i in range(1, n + 1):
        weight, value = items[i - 1]
        for w in range(capacity + 1):
            if weight <= w:
                dp[i][w] = max(dp[i - 1][w], dp[i - 1][w - weight] + value)
            else:
                dp[i][w] = dp[i - 1][w]

        if i % timer_every == 0:
            elapsed_time = time.time() - start_time
            best_solutions_dp.append((elapsed_time, dp[i][-1]))
        if i % print_every == 0:
            print(f"Iteration {i}: Best solution so far for DP: value={dp[i][-1]}, elapsed_time={elapsed_time:.2f}s")

    selected_items = []
    i, w = n, capacity
    while i > 0 and w > 0:
        weight, value = items[i - 1]
        if dp[i][w] != dp[i - 1][w]:
            selected_items.append(i - 1)
            w -= weight
        i -= 1

    return dp[n][capacity], selected_items, best_solutions_dp
```

Listing 3. Differentiable Dynamic Programming Knapsack Algorithm

```python
from collections import namedtuple

Solution = namedtuple("Solution", ["value", "items", "selection", "elapsed_time"])

def soft_knapsack(items, capacity, n_best=10000, iterations=10000, learning_rate=100, temperature=0.5, print_every=500, penalty_factor=1e7):
    n_best=iterations
    n = len(items)
    weights, values = zip(*items)
    weights = torch.tensor(weights, dtype=torch.float)
    values = torch.tensor(values, dtype=torch.float)

    item_selection = torch.rand(n, requires_grad=True)
    optimizer = optim.RMSprop([item_selection], lr=learning_rate)

    best_solutions = []

    start_time = time.time()
    for i in range(iterations):
        optimizer.zero_grad()
        soft_selection = torch.sigmoid(item_selection / temperature)
        total_weight = torch.sum(soft_selection * weights)
        total_value = torch.sum(soft_selection * values)
        capacity_penalty = torch.clamp(total_weight - capacity, min=0) ** 2

        # Multiply capacity_penalty by a large constant
        loss = -(total_value - penalty_factor * capacity_penalty)
        loss.backward()
        optimizer.step()

        # Clamp item_selection values between -5 and 5
        item_selection.data.clamp_(-5, 5)
        if total_weight <= capacity:
          # Update best solutions
          final_selection = torch.sigmoid(item_selection / temperature) > 0.5
          selected_items = [i for i, selected in enumerate(final_selection) if selected]
          max_value = torch.sum(final_selection * values).item()
          current_solution = Solution(max_value, selected_items, final_selection,time.time() - start_time)
          #best_solution = best_solutions[0]
          if len(best_solutions) < n_best:
              best_solutions.append(current_solution)
              best_solutions.sort(key=lambda x: x.value, reverse=True)
          elif max_value > best_solutions[-1].value:
              best_solutions.pop()
              best_solutions.append(current_solution)
              best_solutions.sort(key=lambda x: x.value, reverse=True)
          # Print intermediate results
        if (i + 1) % print_every == 0:
            elapsed_time = time.time() - start_time
            best_solution = best_solutions[0]
            print(f"Iteration {i+1}: loss={loss.item():.2f}, total_value={total_value.item():.2f}, total_weight={total_weight.item():.2f}, elapsed_time={elapsed_time:.2f}s")
            print(f"Best solution so far: value={best_solution.value}, items={best_solution.items}")

    max_value = best_solutions[0].value
    selected_items = best_solutions[0].items

    best_solutions_diff = sorted([(s.elapsed_time, s.value) for s in best_solutions], key=lambda x: x[0])
    #best_solutions_diff = [(s.elapsed_time, s.value) for s in best_solutions]

    return max_value, selected_items, best_solutions_diff
```

APPENDIX I
EXAMPLE CODE FOR PARALLEL RANDOM INITIALIZATION SIMULATION PROGRAM

Listing 4. Example Code for Parallel Random Initialization Simulation Program

```
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader
from torchvision import datasets, transforms
import numpy as np
import random
import itertools

import pyDOE2
from scipy.optimize import minimize
from skopt import gp_minimize

import torchvision
import torchvision.transforms as transforms
import torch.nn.functional as F
import matplotlib.pyplot as plt
import os
from tqdm import tqdm
import datetime
#from datetime import datetime
import csv
from datetime import datetime

results_dir = './results'
def nowTimetoString():
    now=datetime.now()
    return now.strftime("%m-%d-%Y_%H-%M-%S")

class SimpleNN(nn.Module):
    def __init__(self, to_demo=True):
        super(SimpleNN, self).__init__()
        if to_demo:
            self.conv1 = nn.Conv2d(3, 16, 3, padding=1)
            self.conv2 = nn.Conv2d(16, 32, 3, padding=1)
            self.fc1 = nn.Linear(32 * 8 * 8, 64)
            self.fc2 = nn.Linear(64, 10)
        else:
            self.conv1 = nn.Conv2d(3, 16, 3, padding=1)
            self.conv2 = nn.Conv2d(16, 32, 3, padding=1)
            self.fc1 = nn.Linear(32 * 8 * 8, 64)
            self.fc2 = nn.Linear(64, 10)

        self.relu1 = nn.ReLU(inplace=True)
        self.pool = nn.MaxPool2d(2, 2)
        self.relu2 = nn.ReLU(inplace=True)
        self.relu3 = nn.ReLU(inplace=True)

    def forward(self, x):
        x = self.conv1(x)
        x = self.relu1(x)
        x = self.pool(x)
        x = self.conv2(x)
        x = self.relu2(x)
        x = self.pool(x)
        x = x.view(x.size(0), -1)
        x = self.fc1(x)
        x = self.relu3(x)
        x = self.fc2(x)
        return x

def init_weights(model, init_values):
    for i, param in enumerate(model.parameters()):
        param.data.copy_(torch.from_numpy(init_values[i]))

        #param.data.copy_(torch.tensor(init_values[i]))


# SimpleNN and init_weights remain the same as before
def simulate_parallel_loss(model, init_values, train_loader, device, num_epochs=10):
    model.to(device)
    init_weights(model, init_values)
    criterion = nn.CrossEntropyLoss()
    optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.9)

    loss_curve = []
    for epoch in tqdm(range(num_epochs), desc="Training"):#range(num_epochs):
        running_loss = 0.0
        for data, target in train_loader:
            data, target = data.to(device), target.to(device)
            optimizer.zero_grad()
            output = model(data)
            loss = criterion(output, target)
            loss.backward()
            optimizer.step()
            running_loss += loss.item()

        loss_curve.append(running_loss / len(train_loader))
    return loss_curve


#originally work version
def run_simulation(num_workers,init_values, train_loader, device, num_epochs=10, to_demo=True):#original 2nd argument: sampling_method
    losses = []
    for worker_id in range(num_workers):
        #init_values = sampling_method()
        loss_curve = simulate_parallel_loss(SimpleNN(to_demo=to_demo), init_values, train_loader, device, num_epochs)
        losses.append(loss_curve)
    return losses
```

```python
def to_csv(data,num_epochs):
    if not os.path.exists(results_dir):
        os.makedirs(results_dir)
    nowTimeString=nowTimetoString()

    with open(os.path.join(results_dir,
        nowTimeString+'table1.csv'), 'w',
        newline='') as file:
        writer = csv.writer(file)
        header = ['Method']#, 'Iteration
            1', 'Iteration 2', 'Iteration
            3', 'Iteration 4', 'Iteration
            5', 'Iteration 6', 'Iteration
            7', 'Iteration 8', 'Iteration
            9', 'Iteration 10', 'Iteration
            11', 'Iteration 12', 'Iteration
             13', 'Iteration 14', '
            Iteration 15', 'Iteration 16',
            'Iteration 17', 'Iteration 18',
             'Iteration 19', 'Iteration
            20', 'Iteration 21', 'Iteration
             22', 'Iteration 23', '
            Iteration 24', 'Iteration 25',
            'Iteration 26', 'Iteration 27',
             'Iteration 28', 'Iteration
            29', 'Iteration 30']
        for i in range(num_epochs):
            header.append("Epoch"+str(i+1))

        writer.writerow(header)

        for method in data:
            for i in range(len(data[method
                ])):
                row = [method+" worker"+str
                    (i+1)]
                row += [str(num) for num in
                     data[method][i]]
                writer.writerow(row)


def main(to_demo=True,NUM_WORKERS_=20,
    EPOCHS=100,DEMO_EPOCHS=50):
        # Prepare the dataset
    device = torch.device('cuda' if torch.
        cuda.is_available() else 'cpu')
    #device = torch.device('cuda:0' if
        torch.cuda.is_available() else 'cpu
        ')
    #transform = transforms.Compose([
        transforms.ToTensor(), transforms.
        Normalize((0.5, 0.5, 0.5), (0.5,
        0.5, 0.5))])
    transform = transforms.Compose([
        transforms.RandomHorizontalFlip(),
        transforms.RandomCrop(32, padding
            =4),
        transforms.ToTensor(),
        transforms.Normalize((0.5, 0.5,
            0.5), (0.5, 0.5, 0.5))
    ])
    train_dataset = torchvision.datasets.
        CIFAR10(root='./data', train=True,
        download=True, transform=transform)
    if to_demo:
        train_dataset, _ = torch.utils.data
            .random_split(train_dataset,
                [1000, len(train_dataset) -
                1000])
    train_loader = DataLoader(train_dataset
        , batch_size=32, shuffle=True,
        num_workers=2)#, num_workers=2
    #NUM_WORKERS_= 4
    num_workers = NUM_WORKERS_

    num_epochs = DEMO_EPOCHS if to_demo
        else EPOCHS #30
    # Define sampling methods
    def single_random_initialization():
        model = SimpleNN()
        init_values = [p.data.clone().numpy
            () for p in model.parameters()]
            # originally no .numpy()
        return init_values


    def uniform_sampling():
        #return [np.random.uniform(0, 1, p.
            numel()).reshape(p.shape) for p
             in SimpleNN().parameters()]
        model = SimpleNN()
        init_values = [p.data.clone().numpy
            () for p in model.parameters()]
            # originally no .numpy()
        return init_values

    def uniform_init():
        model = SimpleNN()
        init_values = [p.data.clone().numpy
            () for p in model.parameters()]
            # originally no .numpy()
        return nn.init.uniform_(init_values
            )

    def latin_hypercube_sampling():
        n_params = sum(p.numel() for p in
            SimpleNN().parameters())
        lhs_samples = pyDOE2.lhs(n_params,
            samples=num_workers, criterion=
            'maximin')
        lhs_samples = lhs_samples * 2 - 1
            # scale to [-1, 1]

        init_values_list = []
        for sample in lhs_samples:
            init_values = []
            start_index = 0
            for p in SimpleNN().parameters
                ():
                end_index = start_index + p
                    .numel()
                param_sample = sample[
                    start_index:end_index].
                    reshape(p.shape)
                init_values.append(torch.
                    from_numpy(param_sample
                    ))
                start_index = end_index
            init_values_list.append(
                init_values)

        return init_values_list


```

```
    def adaptive_sampling():
        def loss_function(params):
            init_values = [param.reshape(p.shape) for param, p in zip(params, SimpleNN().parameters())]
            return simulate_parallel_loss(SimpleNN(), init_values, train_loader, device)

        bounds = [(-1, 1)] * sum(p.numel() for p in SimpleNN().parameters())
        result = minimize(loss_function, x0=np.zeros(len(bounds)), bounds=bounds, method='L-BFGS-B')
        best_params = result.x

        return [best_params.reshape(p.shape) for p in SimpleNN().parameters()]

    def bayesian_optimization():
        def loss_function(params):
            init_values = [param.reshape(p.shape) for param, p in zip(params, SimpleNN().parameters())]
            return simulate_parallel_loss(SimpleNN(), init_values, train_loader, device)

        bounds = [(-1, 1)] * sum(p.numel() for p in SimpleNN().parameters())
        result = gp_minimize(loss_function, bounds, n_calls=num_workers, n_random_starts=0, random_state=42)
        best_params = result.x

        return [best_params.reshape(p.shape) for p in SimpleNN().parameters()]

    methods = [
        ('Single Random Initialization', single_random_initialization),
        ('Uniform Sampling', uniform_sampling),
        #('Adaptive Sampling', adaptive_sampling),
        #('Bayesian Optimization', bayesian_optimization),
        #('LHS', latin_hypercube_sampling),
    ]

    losses = {}
    for method_name, method in methods:
        print(f"Running {method_name}...")
        if method_name=='Single Random Initialization':
            num_workers=1
        else:
            num_workers=NUM_WORKERS_
        init_values = method()
        loss_curve = run_simulation(num_workers, init_values, train_loader, device, num_epochs,to_demo)
        losses[method_name] = loss_curve

    print(losses)
    to_csv(losses,num_epochs)
    # Plot loss curves
    plt.figure(figsize=(12, 6))
    for method_name, loss_curve in losses.items():
        plt.plot(loss_curve, label=method_name)

    plt.xlabel('Epoch')
    plt.ylabel('Loss')
    plt.title('Loss Curves for Different Initialization Methods')
    plt.legend()
    plt.grid()
    plt.show()

    results_dir = './results'
    if not os.path.exists(results_dir):
        os.makedirs(results_dir)

    now=datetime.now()
    plt.savefig(os.path.join(results_dir, now.strftime("%m-%d-%Y_%H-%M-%S")+'loss_curves.png'))
    plt.show()


if __name__ == "__main__":
    main(to_demo=True,NUM_WORKERS_=40, EPOCHS=50,DEMO_EPOCHS=300)
```