ParallelNAS: A Parallel and Distributed System for Neural Architecture Search

Xiaoyang Qu
Department of Federated Learning
Ping An Technology (Shenzhen) Co., Ltd
Shenzhen, China
quxiaoy@gmail.com

Jianzong Wang
Department of Federated Learning
Ping An Technology (Shenzhen) Co., Ltd
Shenzhen, China
jzwang@188.com

Wenqi Wu
Department of Federated Learning
Ping An Technology (Shenzhen) Co., Ltd
Shenzhen, China
wuwenq7@gmail.com

Jing Xiao
Ping An Technology (Shenzhen) Co., Ltd
Ping An Insurance(Group) Company of China, Ltd
Shenzhen, China
xiaojing661@pingan.com.cn

***Abstract*—Although deep learning takes researchers out of complicated feature engineering, the designing of practical deep learning models is still a laborious process. Neural Architecture Search (NAS) methods have become famous for automating the design of deep learning models. However, NAS methods are very time-consuming. To reduce the search time of NAS methods, we designed a two-level hierarchical parallel system called ParallelNAS, comprising a parallel explorer and parallel evaluators. For the parallel explorer, a general-purposed distributed search framework is built on virtualized, massively-parallel, asynchronous infrastructure. For parallel evaluators, we integrate a segment-balanced sparsity scheme into a mainstream data-parallel architecture. We also design a resource-aware intelligent scheduling algorithm to allocate independent tasks to available virtual workers fully-automatically. The comprehensive experiment results demonstrate our system, with a parallel search policy and parallel evaluators, can reach near-linear speedups on a cluster of GPUs.**

***Index Terms*—Deep Learning, Neural Architecture Search, Parallel Computing**

I. Introduction

In the past few years, deep learning has achieved remarkable successes in various applications. Although deep learning takes researchers and engineers out of complicated feature engineering, the current approach to designing a practical deep learning model is a laborious process. Developing a practical deep learning model requires good intuition, extensive experiments, laborious trials, domain expertise, theoretical demonstration.

Neural Architecture Search(NAS) [50] [2] methods can relieve human effort by automating the design of deep learning models. Neural architecture search can simultaneously learn the network structure and parameters of a deep learning model given specific tasks and constraints. This technique can discover an outstanding deep model outperforming the-state-of-the hand-crafted deep models.

*corresponding author:Jianzong Wang(email:jzwang@188.com)

As shown in Figure 1, the general procedure of NAS methods includes: defining a *search space*, designing an *explorer* (search policy) and designing an *evaluator* (evaluation policy). The explorer select at least one promising candidate architecture from predefined search space. Then, the explorer sends candidate architectures to the evaluator and receives performance estimates of candidate architectures from the evaluator.

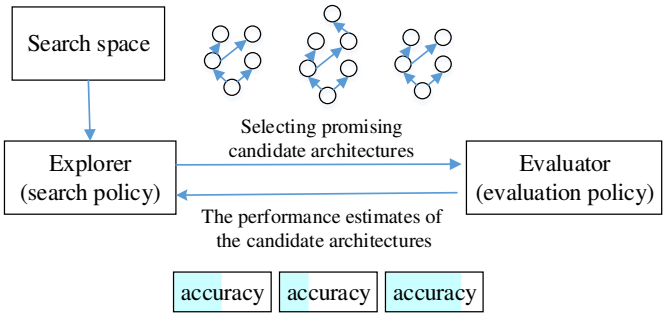


Fig. 1: The diagram of Neural Architecture Search(NAS).

The fundamental problem of NAS methods is time-consuming. For example, NASNet [51] employed 450 GPUs for 3-4 days to discover an outstanding architecture that outperforms a large number of manually designed models for image prediction. Typically, the search space contains billions of architectures, resulting in large-scale search optimization problems. Besides, expensive evaluator accounts for the majority time and resource costs, because it involves training from scratch or retraining.

Researchers have explored a wide range of optimization paradigms to reduce the time cost of NAS methods. There are two major ways to reduce the total search time. One way is to reduce the search iterations, reinforcement learning [50], evolutionary algorithm [34], and Bayes optimization [23] are used to avoid unnecessary iterations for NAS. The

other way is to reduce the computing time in each iteration. Weight sharing [33] and HyperNetwork [5] achieve this goal by reusing parameters.

Parallel computing is an encouraging way to reduce the considerable time consumed by neural network search. As NAS unifies network structure learning and parameters learning into one optimization problem, we can use hierarchical parallel architecture to parallelize and accelerate the search. For the parallel explorer, we designed a centralized parallel architecture with a memory-shared to distributes the candidate architectures to multiple evaluators and collect the feedback values from evaluators. For parallel evaluators, we introduce a segment-balanced sparsity scheme into a data-parallel training architecture. We also design a resource-aware self-learning scheduler to allocate independent tasks to available virtual workers fully-automatically.

The contributions are shown as follows.

- To reduce the search time of NAS methods, we design a hierarchical parallel system called ParallelNAS, consisting of a *parallel explorer* and *parallel evaluators*.
- For *the parallel explorer*, we design a general-purpose architecture with shared memory for NAS methods according to the characteristic features of NAS method.
- For *parallel evaluators* , we optimize a mainstream data-parallel architecture with a segment-balanced sparsity scheme.
- We realize a fully automatic and resource-aware schedule using a virtualized pipeline and a reinforcement-learning algorithm.
- The experiments results prove that our ParallelNAS can obtain near-optimal speedups. Our 16-GPU parallel workers can accomplish challenging model training tasks that previously took two days in about 3 hours.

The outline of this paper is shown as follows. Section II reviews the background of this work and related works. Section III describes the design of our ParallelNAS. Section IV shows comprehensive experimental results. Section V concludes this paper.

II. BACKGROUND

A. Neural Architecture Search

As the NAS technique unifies the neural network weight learning and neural network structure learning into one single optimization problem, the problem can be formulated as

$$\min_{a\in\Lambda} \min_{w_a} Ł(a, w_a) \tag{1}$$

here, a denotes a neural network, Λ denotes the search space, and Ł represent the loss function. We denote the weights of architecture w_a.

After neural architecture search is proven a powerful ability to automatic the design of deep learning models in the paper [50], there emerges a large number of excellent researches on this topic. We classify the NAS methods into two generations, as shown in Table I. The significant difference between the two generation works is the search cost. The search cost of first-generation NAS methods is expensive, up to hundreds or thousands of GPU days. Although the search cost of second-generation NAS methods is significantly reduced, the model accuracy is not competitive.

TABLE I: Popular Neural Network Search Methods

	category	***Method***	EER (100%)	Para (M)	Cost (GPUdays)
1st	RL-based	NASNET-A [51]	2.65	3.3	1800
	EA-based	AmoebaNet [34]	3.34	3.2	3150
	BO-based	PNAS [23]	3.41	3.2	150
2nd	Weight-sharing	ENAS [33]	2.89	4.6	0.5
	Morphism	NASH [10]	6.5	4.4	0.2
	Differentiable	DARTS [25]	2.83	3.4	4
	HyperNetwork	SMASHv2 [5]	4.03	16	3

The key components of NAS methods include search space, search strategies, and evaluation strategies.

1) Search Space: The search spaces contain a wide variety of architectures. Large-scale search space can capture a diverse set of interesting candidate architectures, but result in a large-scale optimization problem, consuming vast time and resources. A simple search space is the space of the chain-structure neural network, where various operations or blocks are applied sequentially on the input tensor and ends with an output tensor in the final. The mainstream design of search space is to apply a repetitive architecture pattern on cell-based structure [23], which is motivated by hand-crafted architectures consisting of repeated motifs, such as VGG, ResNet, and so on. Some neural network is built using particular structures, such as tree-based structure [31], hierarchical structure [24], GNN-based structure [48]. Most of NAS-aware works are based on discrete search space. DARTS [25] and SNAS [45]convert the discrete search space into a continuous search space to enable gradient-based optimizations. In this way, the time cost of the search is substantially reduced.

2) Search Strategies: Once the search space is defined, the explorer operates on the search space. Researchers have explored a wide range of optimization paradigms.

Simple Search. Simple search methods are referred to as the search method without sophisticated exploration mechanisms, such as grid search, random search, local search. Although random search frequently yields high variance, it can surpass many heuristic search policy. Recent work [35] finds that simple random search policy with uniformly randomly samples outperforms many state-of-the-art NAS methods. With a local search method, HillClimbing [11] can also discovery an outstanding neural network.

RL-based (Reinforcement-Learning-based). Reinforcement learning methods [49] [13] are used to train an agent to discover outstanding candidate models using policy gradients. For these methods, the reward is an estimate of the architecture's performance, and the action and the state differs in different RL-based NAS approaches. The first RL-based NAS is proposed in papers [50], which learns an RNN controller that samples a candidate neural network from the search space. To enable the NAS methods scale to a variable-size dataset, NASNET [51] introduces transferring learning to NAS

method.

EA-based (Evolutionary-Algorithm-based). As an alternative to using reinforcement learning, evolutionary techniques evolve a population of candidate models, in every evolution step, with predefined selection algorithm and mutation operations, at least one outstanding model from the population is selected to generate off-springs. For these methods, fitness is an estimate of the architecture's performance, and the selection algorithm and mutation operations differ in different EA-based NAS approaches. Three decades ago, evolutionary algorithm was already used to evolve neural architectures and their weightsas depicted in NEAT [38]. In NEAT, the topology of a neural network is simultaneously evolved along with its weights a hyper parameters. Recent works include CoDeepNEAT and EvolvingDNN [28], which can evolve into promising convolution neural networks. GeneticCNN [44] uses a simple genetic algorithm with binary network representation to discover outstanding architectures. Other evolutionary algorithm , such as genetic programming [40], regularized evolution [34], memetic evolution [26].

In addition, Bayes Optimization has been applied to neural architecture search, such as [31].

3) Evaluation Strategies: The evaluation is computationally expensive because the evaluation requires high-costly model training. In early work, such as [50] [2] [34], the evaluation use a direct estimation(full training). For every candidate model, the evaluator always trains the model from scratch. This straightforward method is easy but wastes too many computational resources and exploration effort. Thereby, researchers have proposed a large number of optimized methods.

Partial Training. The simple idea is to convert the fully training to partial training, such as less training time, less training iterations(early stop), training on a subset of validation dataset (sub-sampling), training on low-resolution images.

Weight sharing. With weight-sharing, the computational resources required for NAS are dramatically reduced from thousands to a few GPU days. Such as ENAS [33]. ENAS [33] learns an RNN controller that samples architectures from the search space and trains the one-shot model based on approximate gradients obtained through reinforcement.

Network morphing. DART [10] can initialize the network with weights from other networks. PNAS [23] use an incremental manner to discover an outstanding network structure.

HyperNetwork. HyperNetwork-based NAS methods samples candidate sub-networks inheriting the weights from the hyper network, such as [5] [3] [14].

B. Parallel Framework

1) Parallel Deep Learning: The mainstream parallel architectures for deep learning include data-parallel and model-parallel architectures. Existing famous deep learning framework, such as MxNet [6], PyTorch [32], TensorFlow [1], Petuum [46], and GeePS [8], support data-parallel training with multiply GPU machine. The representative architectures for data-parallel distributed training contain Parameter Server [19] and AllReduce architecture.

Parameter Server architecture has been used in previous works [1] [6] [7], because its scalable structure in parameter server can distribute model replicas to multiple worker machines. In each iteration, the worker machine, holding a full model replica and partitioned data, compute gradients locally in parallel, send the gradients to a parameter server, then read the averaged gradients from the parameter server for the next iteration.

AllReduce architecture has become popular for deep learning workload because the significant communication costs caused by gradient aggregation can be reduced by pipelining AllReduce operations. As a competitive deep learning framework, PyTorch [32] supports distributed data-parallel training only with the AllReduce architecture. Using the Ring AllReduce algorithm [41], Horovod [36] provides abstraction implementations of efficient AllReduce algorithms and can work as the communication back-end of Tesorflow [1] or MXNet [6].

As the size of the neural network grows, it becomes harder to hide the communication cost of gradient aggregation in Parameter Server or AllReduce architectures. For communication optimization, Parallax [18] uses a sparsity-aware data-parallel training of deep neural networks. For fast data-parallel training, Bosen [43] use a managed communication and consistency, which compromises the strong consistency and weak consistency.

2) Parallel Search Methods: Parallel and distributed schemes are frequently used to reduce the search time on large-scale optimization problems. Here, we reviews some parallel evolution algorithm and parallel reinforcement learning methods.

Parallel Evolution Algorithm. Evolutionary algorithms are frequently used on large-scale optimization problems. There are two options to reduce the overall search time. One option is to reduce the number of search iterations by some optimization algorithms, such as Divide-and-Conquer [47] or Dimension Reduction [4]. The other option is to save the computation time in each iteration using parallel distributed computing . There are two types of distributed model. The first type is "population-distributed" model, which distributes the individuals of population to multiple processors or workers. The representative is master-salve architecture [9]. The other is "dimension-distributed" model distribute separate sub-space to multiple nodes. The representative work is multi-agent [4] and parallel divide-and-conquer based evolutionary algorithm [47].

Parallel Reinforcement Learning. Efforts to parallelize reinforcement learning have been under way for years. A3C [29]use a synchronous SGD strategy to parallelize critic-actor model. Ray [30] is a distributed framework for RL. RLlib [20] provides a scalable RL library. RLPYT [39] is a research code base for deep reinforcement learning in PyTorch. R2D2 [17] achieves a distributed reinforcement learning with recurrent experience replay.

III. Design

A. Architecture of ParallelNAS

The core goal is to reduce the search time of NAS methods, so we build a general-purpose parallel framework for neural architecture search. We develop a two-level hierachical architecture with a virtulized, massively-parallel, asynchronous infrastructure.

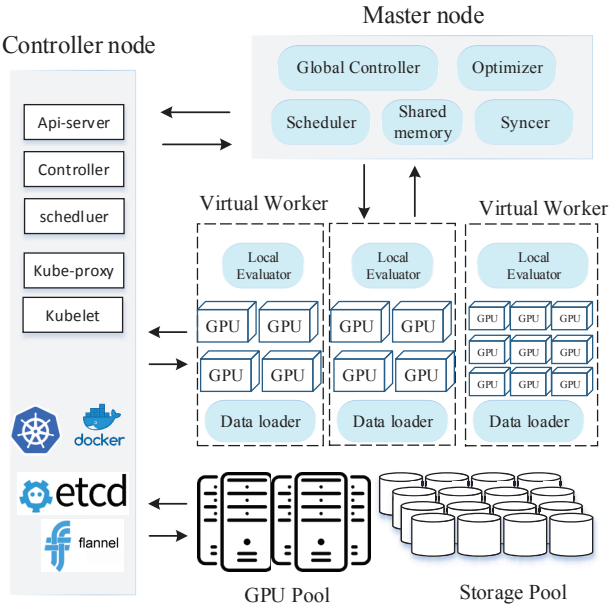


Fig. 2: The Architecture of ParallelNAS. The key components include master node, virtual worker nodes, controller node.

As shown in Figure 2, ParallelNAS consists of *controller node*, *master node*, *virtual worker nodes*.

Master node is mainly in charge of distributing evaluation tasks to multiply evaluators, which operating asynchronously on different virtual worker servers. The details will be illustrated in SectionIII-B. The key components are the shared-memory module shown in SectionIII-C and the automatic and resource-aware scheduling algorithm shown in SectionIII-E.

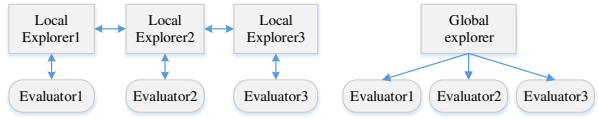
Virtual worker nodes are designed for parallel evaluators. Each evaluator can deploy multiple GPU nodes to implement a data-parallel model training. The details will be depicted SectionIII-B and SectionIII-D. The reason why we call *virtual worker node* rather than *work node* is that the virtual node consists of various numbers of GPU. The number of nodes will be decided by the scheduler, which will be illustrated in SectionIII-E in detail.

Controller node is used for virtualized environment and intelligent scheduling. It consists of various open-source components, such as Docker [27], Kubebernets [16], ETCD, Flannel. For computational reproducibility , Docker [27] packages applications in containers. Kubernetes [16] is a container orchestration system for container management. As the back-end for service discovery, ETCD serves as the orchestration engine for Kubernertes. Flannel is a network fabric for containers, designed for kubernetes and dockers.

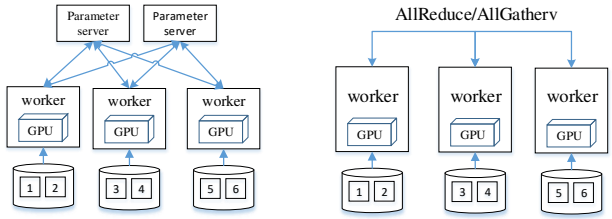
B. Hierarchical Parallel Framework

In this part, we aimed to build a two-level hierarchical parallel framework comprising parallel searching and data-parallel training.

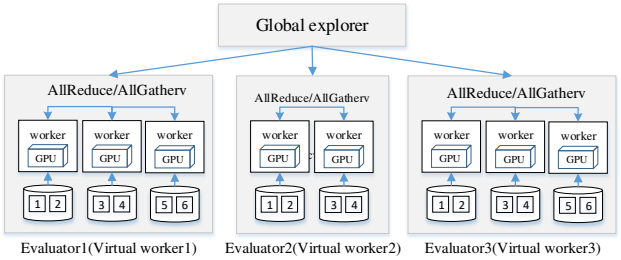
The first-level parallel structure is a distributed framework using a global explorer with multiple evaluators. There are two primary choices for distributed and parallel search. One choice is the decentralized architecture consisting of multiple local explorers and numerous local evaluators, as shown on the left of Figure 3(a). The entire search space is partitioned into various sub search space, so each explorer maintains a separate search space. These local explorers communicate with each other to update their local heuristic function. The other choice is the centralized architecture using a global explorer with multiple local evaluators, as presented on the right of Figure 3(a). The global explore has a global heuristic function and has knowledge of the entire search space. The explorer only distributes the workloads to evaluators and collects feedback value from evaluators. In our NAS problem, it is difficult to partition the search space. As the time consumed by evaluators is vast, the local explorers cannot get enough feedback to update the local heuristic function. Thereby, we use centralized architecture for parallel search.



(a) decentralized and centralized parallel search architecture



(b) Parameter Server architecture and AllReduce architecture



(c) our hierarchical architecture with a parallel explorer and parallel evaluators

Fig. 3: The Impact of Neural Architecture Search Techniques

The second-level parallel framework is a data-parallel training framework for each evaluator with multiple GPUs.The mainstream data-parallel architecture includes Parameter Server architecture and AllReduce architecture. As shown in the left of Figure 3(a), the Parameter Server architecture

collected gradient from multiple workers, and send averaged gradients to various works. The overall dataset is partitioned into several shards and feeds to GPUs in parallel. As shown in the right of Figure 3(b), the AllReduce architecture has no dedicated server to maintain the model parameters. Parameter Server architecture is often bottlenecked on intensive communication due to gradient aggregation. As AllReduce architecture can avoid this problem, it has become the mainstream distributed framework for deep learning. Our parallel architecture for evaluators are developed from AllReduce architecture.

As shown in Figure 3(c), our two-level hierarchical architecture integrates centralized parallel search architecture and AllReduce-based data-parallel training architecture. In the centralized search architecture, the global explorer distributes the workload over multiple virtual workers and collects the feedback value from the multiple virtual workers. Each virtual worker contains multiple GPUs used for data-parallel model training. Thus, AllReduce architecture is used for parallel training in each virtual worker. The centralized architecture is suitable for various large-scale searches, such as evolutional algorithms, reinforcement learning, Bayes optimization, and so on. The AllReduce architecture can be used for any gradient-based model training and model retraining.

Besides, our centralized parallel search also can support the fine-grained search for evaluation without training. Taking the HyperNetwork-based NAS method as an example, the candidate model evaluation does not require model training and retraining. Our centralized parallel search enables multiple independent runs on CPUs rather than GPUs.

C. General-Purpose Parallel Search with Shared Memory

Our parallel search is built on a shared memory with a wait-free communication scheme. For centralized search, virtual workers operate asynchronously on different computing nodes, and the virtual workers do not communicate with each other.

The reason why we use an asynchronous shared memory with a wait-free communication for various NAS methods is shown as follows.

- As illustrated in SectionIII-B, the hierarchical parallel framework provides a fundamental condition for shared memory.
- In order to reduce the expensive computational cost, more and more works reuse the previously validated candidate models for further exploration.
- The model evaluation tasks distributed to multiple evaluators are independent.
- The master-slave architecture with shared memory subsumes all existing search algorithms. Different optimization algorithms differ in the choice of search policy or evaluation policy.

With shared memory, the candidate neural networks generated by NAS methods are stored in shared memory. The shared memory contains directories that represent the candidate neural models. We exploit a wait-free communication. If a virtual worker concurrently updates a candidate model another virtual worker is operating on, the affected virtual worker gives up and tries again after a while. Abandoned candidate models are removed from the shared memory. Namely, abandoned candidate models' directories are periodically garbage-collected.

As shown in Figure 4, the asynchronous shared memory can be applied for various NAS methods. For the RL-based NAS method, an explorer(agent) generates a child model and sends the child model to an available evaluator, then receives the reward of the child model from the corresponding evaluator. The child model is stored in shared memory. For the EA-based NAS method, initially, a population of candidate networks is stored in shared memory. After collecting the fitness from multiple evaluators, the outstanding neural networks are selected to be mutated for new neural networks, and unpromising neural networks are chosen to be removed from the population. Correspondingly, the promising neural networks are written into shared memory, and dropped neural networks are garbage-collected from share memory.

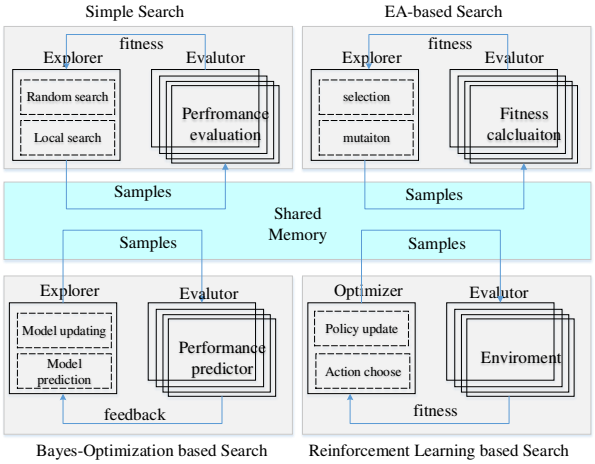


Fig. 4: The general-purpose parallel search architecture with shared memory can be used for various NAS methods.

D. Parallel Approximate Evaluator using Sparsity

As described in Section III-B, we use a two-level hierarchical parallel architecture, and each evaluator can consist of multiple GPUs. The reason why we allocated several GPUs for each evaluator is that the candidate model evaluation requires training from scratch or retraining. For parallel training with AllReduce architecture in each evaluator, we want to optimize the parallelism from the characteristics of AllReduce architecture and NAS methods.

For All-Reduce architecture [41], every gradient aggregation is done using an AllReduce operation, which is similar to the MPI collective operation MPI AllReduce. To aggregate gradients across devices, it executes AllReduce operation and gradient computation operation alternately. In other words, AllReduce operations take place between gradient computation operation.

With the increase of the search iterations, the scale of candidate model parameters dramatically expands, and thereby

it becomes harder to hide the cost of the local gradient computation phase. Sparsity or quantity techniques can be used to reduce the cost of the back-propagation phase. Besides, there is no need for exact training for evaluators. The evaluator can be an approximation evaluator.

Sparsity or quantity techniques not only can reduce the cost of back-propagation but also transmission. A representative work using sparse communication to reduce transmission cost is Deep gradient compression [21]. It introduced a fine-grained sparse communication by setting up a threshold to select important gradients for transmission. Our sparsity-aware AllReduce communication scheme is developed from this idea.

Our sparsity-aware scheme considers some characteristics of AllReduce architecture. The AllReduce is communication-intensive and latency-sensitive, so the synchronization time between clocks depends on the slowest worker. Thus, we proposed a sparsity-aware AllReduce communication scheme with a segment-balanced sparsity scheme for data-parallel architecture.

As shown in Algorithm 1, after gradient computation operation, we set up various thresholds to prune unnecessary gradients for each segment. The ratio of unimportant gradients to be pruned should be the same. AllReduce operations take place after our pruning operation. At the start, every node has a model replica of k segments. It needs 2(k-1) clocks to complete the AllReduce process. Before AllReduce operation starts, every GPU has its locally computed gradients. AllReduce can be into three steps. First, it cost $k-1$ clocks to broadcast the gradients of identical segments to corresponding nodes. Then, the summary operations run for k identical gradients on each node. Lastly, it takes $k-1$ clocks to broadcast the summed gradient for synchronization. After AllReduce operation completes, every GPU node has a full copy of the updated model.

As shown at the end of in Algorithm 1, we also incorporate the surrogate function to stop unpromising samples from reducing the redundant overhead. To pruning out unpromising structures, we can use a rule-based method to identify unpromising models. As an alternative, we use a surrogate function to prune the unpromising network structures.

E. Fully automatic and Resource-aware Scheduler

For every task, we want to build a fully- automatic scheduler. The meaning of a fully automatic scheduler is defined as follows. Giving the specific task and dataset, the only operations executed by the user are defining the search space, choosing the search policy, and choosing performance evaluation strategies. There is no more human participation. The output should be a fully-trained deep learning model that can be directly used for the specific task on the given dataset.

To implement the fully automatic NAS, we build a pipeline for various NAS methods. With the container orchestration tool Kubernetes [16] and shared memory, it is easy to create a pipeline. However, it is difficult to automatically decide how many virtual workers allocated for the task and how many nodes allocated to virtual workers.

Algorithm 1 a sparsity-aware segments-balanced scheme

```
Input: the number of nodes k
Input: the batch size b per node
Input: the data set D_j in node j
while t < T do
  while i < b do
    Sample data x from D_j
    G_t^k = G_t^k + (1/Nb) ∂f(x; w_t)
  end while
  while j < k do
    Select the threshold for j_th segment.
    Remove the unimportant gradients from j_th segments.
  end while
  Take an AllReduce/AllGatherv Operation
  if (t%100) = 0 then
    pred = Performance_prediction()
  end if
end while
```

The optimization is to minimize the total time and maximize resource utilization efficiency. Our scheduler (decision-maker) leverages reinforcement learning for efficient scheduling over the decisions. Here we introduce the detailed setting of the RL-based schedule.

The state space. For each execution of task allocation, we use some features that characterize the sate:

$$(t, \{T^i_{id}, T^i_{type}, T^i_{pro}, E^i_n, E^i_g\}^N_i, G_a) \tag{2}$$

where t is iteration number, $\{\cdot\}^T_i$ represent the N running tasks in current time. T^i_{id} denotes the identifier of the i_{th} task. T^i_{type} denotes the type of task, such as RL-based, EA-based, BO-based, or HyperNetwork-based NAS methods. T^i_{pro} denotes the progress of the i_{th} task, which is calculated like this: the total iteration is divided by the number of the completed iterations. E^i_n and E^i_g respectively denotes the number of evaluators allocated for the i_{th} task and the number of GPUs allocated for each evaluator. G_a and U_i respectively denotes the available GPUs and cluster utilization.

The action space. We use a discrete space of $\{T_e, T_g\}$, where T_e denotes the number of evaluators allocated for the incoming task, and T_g represents the number of GPUs assigned for each evaluator. Note that the available GPU limits the maximum independent evaluator for RL-based or EA-based NAS methods. For various NAS methods, HyperNetwork-based NAS method is a particular case. In this case, the majority of time consumed in hyper-network training and the search process don't require training and re-training. In this case, our fine-grained parallel search policy enables parallel multiple independent runs on CPUs, and our evaluators allow data-parallel training for hyper-network on GPUs.

The *reward* considers two factors: the cluster utilization ratio and speedups. The cluster utilization ratio is calculate as $\sum^M_i \sum^T_j \mathbf{U}^j_i$, where $\mathbf{U}^j_i$ represents the utilization ratio of i_{th} GPU in j_{th} duration.

The *agent* we used is DQN agent. The agent receives the state s_t of the t_{th} iteration from the environment and outputs a discrete action to decide the number of evaluators for the task and the number of GPUs for each evaluator. In DQN, each transition in an episode is (s_t, a_t, R, s_{t+1}), where R is the reward shown after a scheduling period. The objective is formulated as follows:

$$L = \frac{1}{N}\sum_{i}^{N}(y_i - Q(s_i, a_i|\theta^Q))^2 \tag{3}$$

where L denote the loss value, and y_i is predicted quality value. $Q(\cdot)$ represents the estimated quality value.

IV. Experiment

This paper arises with several questions. First, how the NAS methods behave from the parallel search policy compared to the serial search policy? Second, how is the parallelism of the evaluator of ParallelNAS in practice? Lastly, does the model automatically discovered by NAS techniques can outperform the manually-designed model?

A. Experimental Setup and Dataset

To evaluate our system, we set up a cluster with 15 GPU servers, 2 management servers and 7 storage servers. Among 15 GPU servers each with 2 Tesla V100, 2 XEON Gold 6130 CPU, 256 GB memory and 240GB SSD.

The two management servers of E5-2680 each consist of one 4-core CPU, one 256GB memory, and 480GB SSD. The 7 storage servers is of E5-2603 each consist of one 4-core CPU, one 32GB memory, and 8TB HDD storage. These servers are connected by a 10Gb optical Ethernet switch.

As shown in Table IV, we use four datasets from NLP and speech applications. For the VCTK dataset, there are 109 speakers with 44527 utterances in total. We split the overall dataset into a training dataset of 90 speakers and a test dataset of 19 speakers. For each speaker, 10 utterances are randomly chosen as enrollment utterances and another 10 randomly chosen utterances are used as evaluation samples. The real-world large-scale audios set which is collected by our team includes 4000 speakers with 23573 utterances. The ATIS (Airline Travel Information System) and Snips corpus which has been used extensively for intent prediction and slot filling.

B. The Impact of Parallel Search

How the NAS methods behave from the parallel search policy compared to the serial model? To answer this question, we conduct the experiments of two representative NAS methods on various tasks. The experiment results demonstrate that near-optimal speedups can be obtained by our ParallelNAS with up to 16 parallel GPU workers.

Case 1. We choose a representative NAS method from first-generation methods, such as RL-based [50], EA-based [34], and so on. In this case, the evaluator requires full training or partial training.

The search time depends on the iterations number and each iteration step time. As shown in Table II, the iteration number is fixed to 100. The speedup of configuration C_1 over configuration C_2 is defined as the total time consumed by C_2 divided by the time consumed of C_1. To make an accurate evaluation for the speedups, we set fixed iterations time of gradient descent. Namely, there is no need to consider the converge of the deep learning model.

For our search policy, there are two important factors: the number M of independent evaluators allocated for the task and the number N of GPUs for each evaluator. As shown in Table II, the 16 workers each with 1 GPUs have higher parallelism than 1 workers with 16 GPUs, this is because there are no obvious bottleneck for multiple independent runs for the configuration of the 16 parallel workers each with 1 GPUs, but the gradient aggregation decrease the parallelism for the setting of the 1 parallel workers each with 16 GPUs, In addition, this results suggest that with more independent evaluators with more GPUs, we can achieve higher speedups.

Case 2. We choose a representative NAS method from second-generation NAS methods, such as weight-sharing [33], hyper-network [5], network morphing [10]. The parameters are frequently reused in these methods.

The representative NAS method chosen to evaluate our parallel search is a HyperNetwork-based NAS method, namely, a one-shot model architecture search on hyper network. This method is implemented in three steps. Firstly, we should train the hyper network. Second, discovering the best-accuracy sub-network by random search or other heuristic search functions. Thirdly, retraining the best-accuracy sub-network. In this kind of

As shown in Table III, we evaluate our parallel search on two tasks on two datasets. For voice tasks on the VCTK dataset, our 16-GPU parallel workers can accomplish these challenging tasks that previously took 46.82 hours in 3.77 hours. Namely, it achieves up to 12.4x speedups. With smaller model, we can achieve a higher speedup. For the NLP task on the ATIS dataset, the 16-GPU parallel workers can achieve up to 11.88 speedups.

C. The Impact of Parallel Evaluator

To evaluate the parallel efficiency of our parallel evaluator, we make a comparison with Parameter Server architecture and AllReduce architecture. We train the hyper-network on Parameter Server architecture implemented using the original distributed components of *Tensorflow* [1], and on AllReduce architecture using the *Horovod* [36] tool.

The speedup of configuration C_1 over configuration C_2 is defined as the total time consumed by C_2 divided by the time consumed of C_1.

As shown in Figure 5, the training speed of our parallel evaluator for the hyper-network model is faster than Parameter Server architecture and AllReduce architecture. While the speedups of Parameter Server architectures and AllReduce are 6.1x and 12.2x, our parallel evaluator achieves up to 13.2x. The parameter of hypernetwork is huge, so the communication

TABLE II: The speedups on RL-based or EA-based Neural Network Search.The speedup of configuration C_1 over configuration C_2 is defined as the tolal time consumed by C_2 divided by the time consumed of C_1.

	Task Type	*Iterations*	*1 worker(16GPU)*	*4 worker(4GPU)*	*8 worker(2GPU)*	*16 worker(1GPU)*
RL-based or EA-based	NLP	1x	13.9x	14x	14.2x	15.2x
	Voice	1x	12x	13.6x	13.4x	15.1x

TABLE III: The search time of Hyper-Network based NAS with various distribute configuration on a voice task and a NLP task

	Type	*Parameters*	*1 worker(1GPU)*	*1 worker(4GPU)*	*1 worker(8GPU)*	*1 worker(16GPU)*
Voice task on VCTK	small	2.43M	14.6h	3.99h	3.06h	1.23h
	medium	17.04M	33.9h	9.52h	6.23h	2.54h
	large	46.82M	50.7h	14.06h	9.17h	3.77h
NLP task on ATIS	small	0.26M	0.74h	0.22h	0.12h	0.06h
	medium	0.49M	1.55h	0.42h	0.25h	0.13h
	large	0.95M	3.09h	0.84h	0.51h	0.25h

TABLE IV: The speedups on weights-sharing Neural Network Search

	Dataset	*category*	*Train*	*Val*	*Test*
voice Task	VCTK	109	1800	-	190
	Lage-scale	4000	39600	-	2850
NLP Task	ATIS	21	4478	500	893
	SNIP	7	13084	700	700

cost of gradient aggregation will slow down the Parameters Server Architecture. As our parallel evaluator integrates the sparsity and segment-balanced into AllReduce architecture, our scheme achieves higher speedups than AllReduce architecture.

As the theoretically optimal speedup is linear speedup, the ideal speedups are 16x with 16 GPUs. As shown in Figure 5, for various number of GPUs, the speedups of our parallel evaluator are always the closest one to the ideal architecture. As the network we trained is very large, there is a gap between ideal architecture and real architecture.

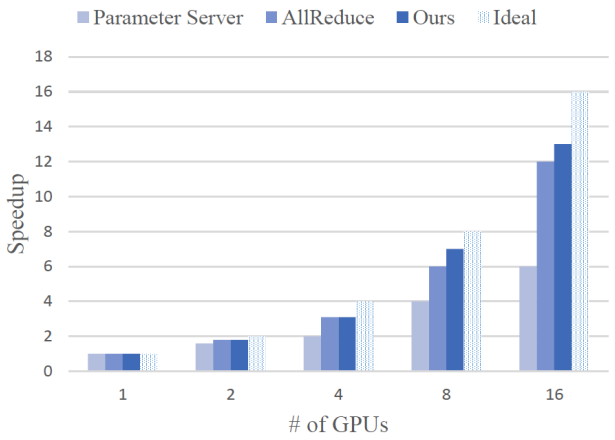


Fig. 5: The speedups comparison of hyper-network training on four distributed architecture: Parameter Server architecture, AllReduce architecture, our parallel evaluator, ideal architecture

D. The Impact of Neural Architecture Search Techniques

The last question is that, does model automatically designed by NAS techniques can outperform the manually-designed model? There are a large number of works [33] [25] that have demonstrated that the NAS-based models outperform expert-designed deep learning model in image application, we conduct experiments emerging applications, such as NLP tasks and voice tasks.

Voice-print verification is a typical task in the field of speech field. The estimation metric of voice-print verification is error equal rate(EER). This metric is to find the threshold extracted by the crossing point between the false-positive curve and the false-negative curve.

As shown in Figure 6, the NAS-base model is compared against two human-designed models: X-vector [37] and LSTM model with a generalized end-to-end loss function [42]. Our NAS-based model implemented based on using the super network, using a cell-based search space and evolutionary search policy.

In overall, our NAS-based model significantly performs better than X-vector [37] and LSTM-GE2E [42]. On VCTK dataset, the equal error rate(EER) of LSTM-GE2E(6.2%) and X-vector(4.6%) are higher than our NAS-based model(1.8%). Similarly, the EER of our NAS-based model obviously lower than the other two models on our large-scale private dataset.

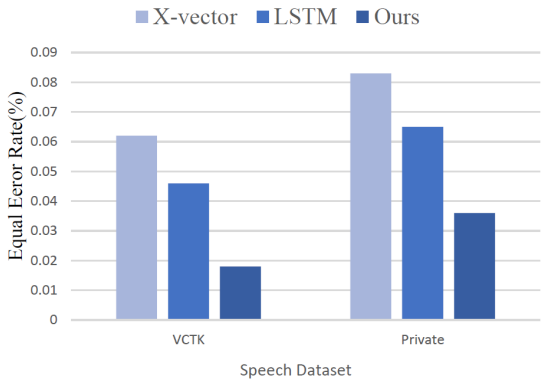
Intent prediction and slot filling problem is an important problem in NLP application. The estimation metric of the Intent prediction is accuracy. The estimation metric of slot filling is F1 score, which is a weighted average of the precision and recall.

As shown in Figure 6, we compare our NAS-based model with three state-of-the-art expert-designed joint models, including using bidirectional RNN-LSTM (RNN-LSTM) [15], Attention-based Bidirectional RNN (Atten-BiRNN) [22], slot-gated attention-based mode (Slot-Gated) [12]. Our NAS-based model for the Intent prediction and slot filling problem is also implemented based on Hyper-Network.

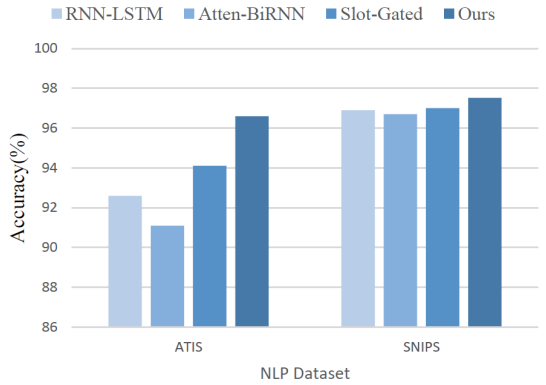
Overall, our NAS-based model can afford to optimize the intent prediction problem to higher accuracy than manually-designed models. On the ATIS dataset, the accuracy of our

approach is higher than state-of-the-art expert-designed models.

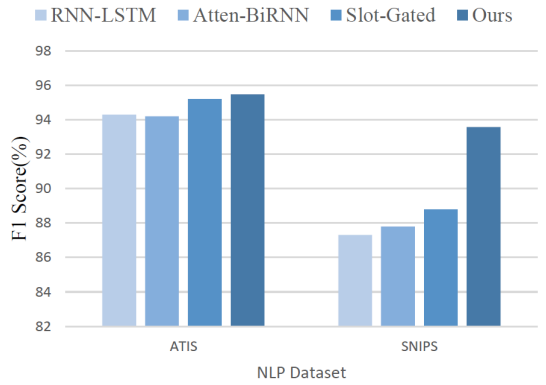
For the slot filling part, our NAS-based model yields a far higher F1 score than we assumed. Especially on the SNIPS dataset, the F1 score of our NAS-based model(93.57%) overwhelmingly outperforms RNN-LSTM(87.3%), Atten-BiRNN(87.8%), Slot-Gate(88.8%). On the ATIS dataset, our method performs slightly better than other methods because the baseline is too high to make a significant improvement.



(a) The Equal Error Rate(EER) comparison of three voice-print verification models on two voice datasets



(b) The accuracy comparison of intent prediction on two NLP datasets



(c) The F1 score comparison of slot filling on two NLP datasets

Fig. 6: The Impact of Neural Architecture Search Techniques

V. Conclusion

In this paper, we designed a parallel and distributed system called ParallelNAS for neural network search. The iterative search framework contains two key components: an explorer (search policy) and an evaluator (sample evaluation). For parallel iterative search, a general-purposed distributed search framework is built on virtualized, massively-parallel, asynchronous infrastructure. As the time and resources consumed by the evaluator account for the vast majority of the proportion, we propose an approximate evaluation policy using a data-parallel architecture with a segment-balanced sparsity policy. And we use a learned surrogate function to avoid unnecessary iterations. We also design an automatic scheduling algorithm aimed to completely automatic the model with maximum resource utilization.

We conduct comprehensive experiments on various applications, such as speech task and NLP task. The results of the experiment demonstrate that our ParallelNAS can obtain near-optimal speedups with a parallel search policy and optimized parallel evaluators. Our 16-GPU parallel workers can accomplish challenging model training tasks that previously took 2 days in about 3 hours. Besides, the experimental results show that the model automatically designed by NAS techniques can outperform expert-designed models.

References

[1] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "Tensorflow: A system for large-scale machine learning," in *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, 2016, pp. 265–283.

[2] B. Baker, O. Gupta, N. Naik, and R. Raskar, "Designing neural network architectures using reinforcement learning," *arXiv preprint arXiv:1611.02167*, 2016.

[3] G. Bender, P.-J. Kindermans, B. Zoph, V. Vasudevan, and Q. Le, "Understanding and simplifying one-shot architecture search," in *International Conference on Machine Learning*, 2018, pp. 549–558.

[4] P. Bouvry, F. Arbab, and F. Seredynski, "Distributed evolutionary optimization, in manifold: Rosenbrock's function case study," *Information Sciences*, vol. 122, no. 2-4, pp. 141–159, 2000.

[5] A. Brock, T. Lim, J. M. Ritchie, and N. Weston, "Smash: one-shot model architecture search through hypernetworks," *arXiv preprint arXiv:1708.05344*, 2017.

[6] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang, "Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems," *arXiv preprint arXiv:1512.01274*, 2015.

[7] T. Chilimbi, Y. Suzue, J. Apacible, and K. Kalyanaraman, "Project adam: Building an efficient and scalable deep learning training system," in *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*, 2014, pp. 571–582.

[8] H. Cui, H. Zhang, G. R. Ganger, P. B. Gibbons, and E. P. Xing, "Geeps: Scalable deep learning on distributed gpus with a gpu-specialized parameter server," in *Proceedings of the Eleventh European Conference on Computer Systems*. ACM, 2016, p. 4.

[9] M. Dubreuil, C. Gagné, and M. Parizeau, "Analysis of a master-slave architecture for distributed evolutionary computations," *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, vol. 36, no. 1, pp. 229–235, 2006.

[10] J.-H. M. Elsken, Thomas and F. Hutter., "Simple and efficient architecture search for convolutional neural networks," *arXiv preprint arXiv:1711.04528*, 2017.

[11] T. Elsken, J.-H. Metzen, and F. Hutter, "Simple and efficient architecture search for convolutional neural networks," *arXiv preprint arXiv:1711.04528*, 2017.

[12] C.-W. Goo, G. Gao, Y.-K. Hsu, C.-L. Huo, T.-C. Chen, K.-W. Hsu, and Y.-N. Chen, "Slot-gated modeling for joint slot filling and intent prediction," in *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 2 (Short Papers)*, 2018, pp. 753–757.

[13] M. Guo, Z. Zhong, W. Wu, D. Lin, and J. Yan, "Irlas: Inverse reinforcement learning for architecture search," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2019, pp. 9021–9029.

[14] Z. Guo, X. Zhang, H. Mu, W. Heng, Z. Liu, Y. Wei, and J. Sun, "Single path one-shot neural architecture search with uniform sampling," *arXiv preprint arXiv:1904.00420*, 2019.

[15] D. Hakkani-Tür, G. Tür, A. Celikyilmaz, Y.-N. Chen, J. Gao, L. Deng, and Y.-Y. Wang, "Multi-domain joint semantic frame parsing using bi-directional rnn-lstm." in *Interspeech*, 2016, pp. 715–719.

[16] K. Hightower, B. Burns, and J. Beda, *Kubernetes: up and running: dive into the future of infrastructure*. " O'Reilly Media, Inc.", 2017.

[17] S. Kapturowski, G. Ostrovski, J. Quan, R. Munos, and W. Dabney, "Recurrent experience replay in distributed reinforcement learning," 2018.

[18] S. Kim, G.-I. Yu, H. Park, S. Cho, E. Jeong, H. Ha, S. Lee, J. S. Jeong, and B.-G. Chun, "Parallax: Sparsity-aware data parallel training of deep neural networks," in *Proceedings of the Fourteenth EuroSys Conference 2019*. ACM, 2019, p. 43.

[19] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su, "Scaling distributed machine learning with the parameter server," in *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*, 2014, pp. 583–598.

[20] E. Liang, R. Liaw, R. Nishihara, P. Moritz, R. Fox, J. Gonzalez, K. Goldberg, and I. Stoica, "Ray rllib: A composable and scalable reinforcement learning library," *arXiv preprint arXiv:1712.09381*, 2017.

[21] Y. Lin, S. Han, H. Mao, Y. Wang, and W. J. Dally, "Deep gradient compression: Reducing the communication bandwidth for distributed training," *arXiv preprint arXiv:1712.01887*, 2017.

[22] B. Liu and I. Lane, "Attention-based recurrent neural network models for joint intent detection and slot filling," *arXiv preprint arXiv:1609.01454*, 2016.

[23] C. Liu, B. Zoph, M. Neumann, J. Shlens, W. Hua, L.-J. Li, L. Fei-Fei, A. Yuille, J. Huang, and K. Murphy, "Progressive neural architecture search," in *Proceedings of the European Conference on Computer Vision (ECCV)*, 2018, pp. 19–34.

[24] H. Liu, K. Simonyan, O. Vinyals, C. Fernando, and K. Kavukcuoglu, "Hierarchical representations for efficient architecture search," *arXiv preprint arXiv:1711.00436*, 2017.

[25] H. Liu, K. Simonyan, and Y. Yang, "Darts: Differentiable architecture search," *arXiv preprint arXiv:1806.09055*, 2018.

[26] P. R. Lorenzo and J. Nalepa, "Memetic evolution of deep neural networks," in *Proceedings of the Genetic and Evolutionary Computation Conference*. ACM, 2018, pp. 505–512.

[27] D. Merkel, "Docker: lightweight linux containers for consistent development and deployment," *Linux Journal*, vol. 2014, no. 239, p. 2, 2014.

[28] R. Miikkulainen, J. Liang, E. Meyerson, A. Rawal, D. Fink, O. Francon, B. Raju, H. Shahrzad, A. Navruzyan, N. Duffy *et al.*, "Evolving deep neural networks," in *Artificial Intelligence in the Age of Neural Networks and Brain Computing*. Elsevier, 2019, pp. 293–312.

[29] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning," in *International conference on machine learning*, 2016, pp. 1928–1937.

[30] P. Moritz, R. Nishihara, S. Wang, A. Tumanov, R. Liaw, E. Liang, M. Elibol, Z. Yang, W. Paul, M. I. Jordan *et al.*, "Ray: A distributed framework for emerging {AI} applications," in *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, 2018, pp. 561–577.

[31] R. Negrinho and G. Gordon, "Deeparchitect: Automatically designing and training deep architectures," *arXiv preprint arXiv:1704.08792*, 2017.

[32] A. Paszke, S. Gross, S. Chintala, and G. Chanan, "Pytorch: Tensors and dynamic neural networks in python with strong gpu acceleration," *PyTorch: Tensors and dynamic neural networks in Python with strong GPU acceleration*, vol. 6, 2017.

[33] H. Pham, M. Y. Guan, B. Zoph, Q. V. Le, and J. Dean, "Efficient neural architecture search via parameter sharing," *arXiv preprint arXiv:1802.03268*, 2018.

[34] E. Real, A. Aggarwal, Y. Huang, and Q. V. Le, "Regularized evolution for image classifier architecture search," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 33, 2019, pp. 4780–4789.

[35] C. Sciuto, K. Yu, M. Jaggi, C. Musat, and M. Salzmann, "Evaluating the search phase of neural architecture search," *arXiv preprint arXiv:1902.08142*, 2019.

[36] A. Sergeev and M. Del Balso, "Horovod: fast and easy distributed deep learning in tensorflow," *arXiv preprint arXiv:1802.05799*, 2018.

[37] D. Snyder, D. Garcia-Romero, G. Sell, D. Povey, and S. Khudanpur, "X-vectors: Robust dnn embeddings for speaker recognition," in *2018 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 2018, pp. 5329–5333.

[38] K. O. Stanley and R. Miikkulainen, "Evolving neural networks through augmenting topologies," *Evolutionary computation*, vol. 10, no. 2, pp. 99–127, 2002.

[39] A. Stooke and P. Abbeel, "rlpyt: A research code base for deep reinforcement learning in pytorch," *arXiv preprint arXiv:1909.01500*, 2019.

[40] M. Suganuma, S. Shirakawa, and T. Nagao, "A genetic programming approach to designing convolutional neural network architectures," in *Proceedings of the Genetic and Evolutionary Computation Conference*. ACM, 2017, pp. 497–504.

[41] R. Thakur, R. Rabenseifner, and W. Gropp, "Optimization of collective communication operations in mpich," *The International Journal of High Performance Computing Applications*, vol. 19, no. 1, pp. 49–66, 2005.

[42] L. Wan, Q. Wang, A. Papir, and I. L. Moreno, "Generalized end-to-end loss for speaker verification," in *2018 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 2018, pp. 4879–4883.

[43] J. Wei, W. Dai, A. Qiao, Q. Ho, H. Cui, G. R. Ganger, P. B. Gibbons, G. A. Gibson, and E. P. Xing, "Managed communication and consistency for fast data-parallel iterative analytics," in *Proceedings of the Sixth ACM Symposium on Cloud Computing*. ACM, 2015, pp. 381–394.

[44] L. Xie and A. Yuille, "Genetic cnn," in *Proceedings of the IEEE International Conference on Computer Vision*, 2017, pp. 1379–1388.

[45] S. Xie, H. Zheng, C. Liu, and L. Lin, "Snas: stochastic neural architecture search," *arXiv preprint arXiv:1812.09926*, 2018.

[46] E. P. Xing, Q. Ho, W. Dai, J. K. Kim, J. Wei, S. Lee, X. Zheng, P. Xie, A. Kumar, and Y. Yu, "Petuum: A new platform for distributed machine learning on big data," *IEEE Transactions on Big Data*, vol. 1, no. 2, pp. 49–67, 2015.

[47] P. Yang, K. Tang, and X. Yao, "A parallel divide-and-conquer based evolutionary algorithm for large-scale optimization," *arXiv preprint arXiv:1812.02500*, 2018.

[48] C. Zhang, M. Ren, and R. Urtasun, "Graph hypernetworks for neural architecture search," *arXiv preprint arXiv:1810.05749*, 2018.

[49] Z. Zhong, J. Yan, and C.-L. Liu, "Practical network blocks design with q-learning," *arXiv preprint arXiv:1708.05552*, vol. 1, no. 2, p. 5, 2017.

[50] B. Zoph and Q. V. Le, "Neural architecture search with reinforcement learning," *arXiv preprint arXiv:1611.01578*, 2016.

[51] B. Zoph, V. Vasudevan, J. Shlens, and Q. V. Le, "Learning transferable architectures for scalable image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2018, pp. 8697–8710.