Retiarii: A Deep Learning Exploratory-Training Framework

Quanlu Zhang, Zhenhua Han, Fan Yang, Yuge Zhang, Zhe Liu, Mao Yang, and Lidong Zhou, *Microsoft Research*

https://www.usenix.org/conference/osdi20/presentation/zhang-quanlu

This paper is included in the Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation

November 4–6, 2020

978-1-939133-19-9

Open access to the Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation is sponsored by USENIX

Retiarii: A Deep Learning Exploratory-Training Framework

Quanlu Zhang, Zhenhua Han, Fan Yang, Yuge Zhang, Zhe Liu, Mao Yang, Lidong Zhou

Microsoft Research

Abstract

Traditional deep learning frameworks such as TensorFlow and PyTorch support training on a single deep neural network (DNN) model, which involves computing the weights iteratively for the DNN model. Designing a DNN model for a task remains an experimental science and is typically a practice of deep learning model exploration, dovetailed with training and validation, aiming to find the best model among a set that yields the best result. Retrofitting such exploratory-training into the training process of a single DNN model, as supported by current deep learning frameworks, is unintuitive, cumbersome, and inefficient, because of the fundamental mismatch between exploring a set of models and training a single one.

Retiarii is the first framework to support deep learning exploratory-training. In particular, Retiarii (i) provides a new programming interface to specify a DNN model space for exploration, as well as an interface to describe the exploration strategy that decides which order to instantiate and train models in, how to prioritize model training, and when to terminate training of certain models; (ii) offers a Just-In-Time (JIT) engine that instantiates models, manages the training of the instantiated models, gathers the information for the exploration strategy to consume, and executes the decisions accordingly; (iii) identifies the correlations between the instantiated models and develops a set of cross-model optimizations to improve the overall exploratory-training process. Retiarii does so by introducing a key abstraction, *Mutator*, that connects the specifications of DNN model spaces and exploration strategies, while exposing the correlations between models for optimization. As a result, Retiarii's clean separation of DNN model space specification, exploration strategy, and cross-model optimizations, connected through the single mutator abstraction, leads to ease of programming, reuse of components, and vastly improved (up to 8.58x) overall exploratory-training efficiency.

1 Introduction

Deep neural networks (DNNs) have been successfully applied to a variety of perception-based tasks such as vision and speech. For each such task, a DNN model architecture, depicted as a graph of operators as vertices, connected with weighted edges, is designed. The model is then trained to populate the weights, before it can be used to perform the task. Deep learning frameworks, such as TensorFlow [11] and PyTorch [48], have been designed to describe an individual DNN model and train the model as a (training) job to run on target hardware, such as GPUs. Training a deep learning model is often resource intensive and costly.

Devising a model for a particular task often involves an iterative exploration process, where a developer would often start with a model architecture that captures the main intuitions and tweak it repeatedly until a model with satisfactory results is identified in a continuous training and validation process. Alternatively, a model architecture could also evolve from simple models following a simple set of evolution rules.

There are clear gaps between the needs to support this *exploratory-training* process and the existing deep learning frameworks. First, this exploratory-training process works on a series of deep learning models, rather than a single one, as supported by the existing deep learning frameworks. A developer either has to specify each model individually in a manual, tedious, and repetitive process, or encodes this series of models as one "jumbo" model [13, 27, 50, 65] using advanced features such as dynamic graph and control flow. Such a "jumbo" model pollutes the original model architecture and makes it significantly harder to understand as changes are scattered across the model description with complex dynamic, control-flow structures. It is also more difficult to optimize due to the use of those dynamic, control-flow structures.

Second, deep learning frameworks manage individual training jobs and cannot capture or leverage the correlation among the set of training jobs in the same exploratory-training process. A developer is again forced to code certain exploration strategies in a "jumbo" model, together with ad hoc runtime mechanisms to manipulate the priorities of jobs or stop not-so-promising jobs early. Such implementations of exploration strategies are hardly reusable as they are deeply coupled with and embedded in a particular exploratory-training process. And there is no easy way to expose the correlations among those models, which tend to share many common structures, for cross-model optimizations. Training a set of models often incurs significant cost; any efficiency gains through optimizations would often allow an exploratory-training process to find a better model under the same budget.

We therefore propose Retiarii, the first deep learning framework specifically designed to support exploratory-training. To address the gaps we have previously identified in the existing deep learning frameworks, we address three core problems of exploratory-training: (i) specifying a DNN model space to explore, (ii) defining and realizing exploration strategies

to decide when to instantiate a model in the space, which ones to instantiate, how to prioritize the training of the instantiated models, and when to terminate the jobs for training those models, and (iii) exposing the correlations among the instantiated models and optimizing training across models by leveraging the correlation information.

Retiarii embraces a new *Mutator* abstraction as the basis for specifying a DNN model space and for defining an exploration strategy. Observing that the exploratory-training process tends to introduce relatively minor modifications to existing models or to compose simple models together following a set of evolution rules, Retiarii allows developers to specify each such modification or evolution as a *mutator* on a model graph. A DNN model space for an exploratory-training process can be defined as a set of base models (each specified as in the original deep learning frameworks, with no "pollution") and a set of mutators. The DNN mode space is then the base models, plus any subsequent models produced by applying mutators to the current models, and so on. An exploration strategy can then be partly defined to govern when to generate new models by applying mutators, as well as which current models and mutators to choose.

Retiarii further designs a Just-In-Time (JIT) engine for the exploratory-training process, which essentially manages the logical collection of all models and their corresponding training jobs. The engine instantiates new models dynamically, exposes the correlations of the instantiated models for cross-model optimizations, schedules the optimized jobs for execution, and manages the execution of the scheduled jobs, governed by the specified exploration strategy.

Retiarii advocates a clean separation of concerns and strives for simplicity and modularity. The mutator abstraction focuses on the changes to an existing model and exposes the differences (and similarities) of models for cross-model optimizations. Each mutator is fine-grained, to capture a logical unit of modification, and intended to be composable and reusable. The cross-model optimizations are also designed and implemented as general capabilities, enabled by the mutator abstraction, in Retiarii's JIT engine. Exploration strategies are decoupled from the specification of the model spaces (through base models and mutators) to maximize reusability, even though some exploration strategies might unavoidably have dependencies on certain types of model spaces.

We have fully implemented and open sourced Retiarii [1]. So far, Retiarii implements 6 mutators to define 18 different model spaces, 11 different exploration strategies, and 3 cross-model optimizations. These combinations have already covered 27 NAS algorithms from the research community, and benefit from vastly improved performance with cross-model optimizations. Our evaluation shows that (1) Retiarii reduces the exploration time of popular Neural Architecture Search (NAS) algorithms by up to 2.57×, and (2) Retiarii improves the scalability of NAS algorithms using weight sharing with a speed-up of 8.58×.

[1] Source code available at https://github.com/microsoft/nni/tree/retiarii_artifact

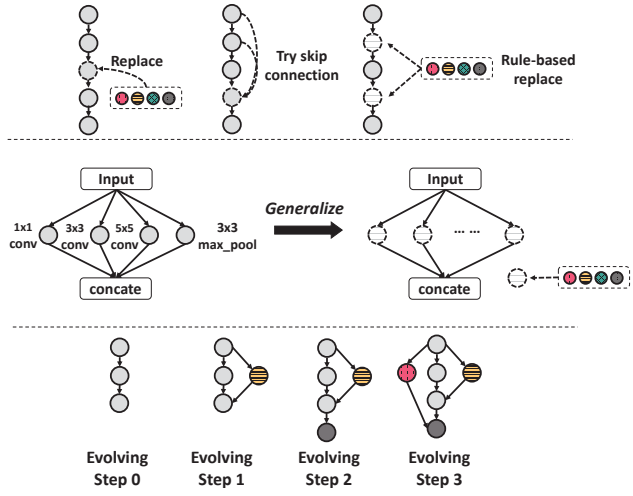


Figure 1: Three typical types of model space explorations.

2 Background and Motivation

The many ways of creating candidate model variations. Developing a model typically involves creating interesting candidate model variations following some design intuitions; for example, by 1) tweaking a substructure (*e.g.,* a layer or a cell) of a base model, 2) coming up with generalized cell structure, 3) or evolving network structure gradually, as shown in Figure 1.

The top set of examples in Figure 1 shows different ways of modifying a base model. One could replace an operator at a layer with some candidate operators (*e.g., normal conv*, *depthwise conv*), or changing a layer's input (*e.g.,* adding some skip connections). The modification can also be applied to a *cell* containing several interconnected layers, but treated as a one logical layer. More generally, a matching rule can be defined to apply modifications on the entire model (*e.g.,* adding *BatchNorm* after convolution layers or replacing all *ShuffleNet* cells [42] with *Inverted Residual* cells [54]).

The middle example in Figure 1 shows how one could generalize a cell structure in order to find a better one. For example, an Inception cell [57] can be generalized to explore a space with different numbers of paths and a different operator on each path. Similarly, an LSTM structure can be generalized to an RNN cell [69]. A generalized structure usually contains a large number of different structures.

The bottom example in Figure 1 shows how the final network gradually evolves from a simple network following some rules. The rules could be adding a layer/edge or changing a layer's operator in each evolution step [23].

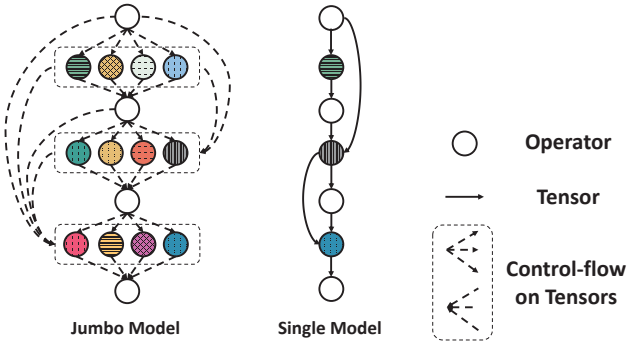


Figure 2: The jumbo model compared to a single model in the model space. Colored circles are different operators.

The pain of specifying and exploring a model space. Exploring a model space, as exemplified in Figure 1, is not directly supported by the existing deep learning frameworks, such as TensorFlow and PyTorch. A model developer often has to program and train each model manually, or to code up all the variations of models in a model space as a single *jumbo model* in TensorFlow/PyTorch through complex control-flow, such as using specified values on the condition of control-flows to route to each model [27, 50, 62, 70]. Figure 2 shows a simple example, a layer has four candidate operators (*e.g.,* *normal conv*, *depthwise conv*, *avgpool*, and *maxpool*), there should be a control-flow to pick one during model construction. If a layer's input is the output of one of the previous layers (*e.g.,* skip connection), there should be a dynamic control-flow to route to the right path during model *forward* (*i.e.,* forward pass of data flow graph). Some popular model spaces [50, 62] change operators and inputs on as many as tens of layers, leading to excessive complexity, making the code hard to understand, and going beyond the limited capabilities of current frameworks to handle control-flow. The control-flow in jumbo models also make them hard to apply compiler optimization techniques, such as operator fusion [15, 38] and memory planning [16]. Figure 3 shows the performance gaps in terms of throughout for ResNet50, as a single model vs. as one encoded as part of a jumbo model.

Automatic model exploration. A DNN model space can be explored automatically with an *exploration strategy*. The action scope of exploration strategy spans from model generation to model execution.

When exploring a huge model space, it is usually impossible and unnecessary to train all the models in the space. An exploration strategy is responsible for deciding which models to instantiate and train, in what priority, and when to terminate. A typical strategy on which models to instantiate could be *brute force* (*e.g.,* random search [56] and grid search [60]), *heuristic-based* (*e.g.,* evolution [23, 30] and annealing algorithms [36]), or more advanced *model-based* (*e.g.,* Bayesian

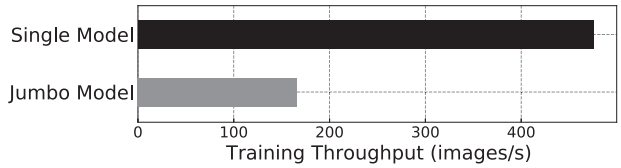


Figure 3: The throughput of ResNet50 built as a single model and a jumbo model. The space contains 4 choices of convolution operator at each layer. Both computation graphs are optimized by TensorFlow XLA [38].

models [33, 67] and reinforcement learning [59, 69, 70]).

An exploration strategy further manages the executions of training instantiated model; for example, to stop the execution of a bad-performing model early based on a performance predictor [20], or to dynamically adjust the computation resource provided to each model depending on the model's performance [64], or to run several mini-batches only and share the weights of overlapped layers among the models to reduce each model's execution time significantly [27, 50].

The pain of implementing exploration strategies. An exploration strategy naturally manages a set of models. Implementing such a strategy with the existing deep learning frameworks is unintuitive and cumbersome, as those frameworks are designed for training individual models and have no support for an exploration strategy.

Because an exploration strategy intensively involves instantiating models from a model space, the implementation often tightly couples an exploration strategy with a specific model space, further increasing the complexity of already complicated jumbo models. For example, an RNN-based RL algorithm (a popular exploration strategy) uses each of its time steps to control the condition value of each control-flow in the jumbo model [50]. Further incorporating the logic of controlling model training makes the jumbo model unmanageable. As a result, though most exploration strategies are logically applicable to different model spaces, the implementations embedded in the jumbo models are hardly reusable by other model spaces.

Encoding an exploration strategy in a jumbo model also makes it hard to expose cross-model optimization opportunities as an exploratory-training usually produces many models. The models explored tend to have strong correlations (*e.g.,* common computation logic) among them, as the variations produced tend to touch only a certain part of the model, while keeping the rest unchanged. The training of those models also share the same dataset and data preprocessing logic. To adapt a model to different tasks, the large backbone network (*e.g.,* BERT) is often fixed: the exploration tends to focus on varying the structure of several added layers. Significant opportunities, therefore, exist in leveraging common computation across model training to speed up an exploratory-training process as a whole. When encoding an exploration strategy in

这里和 finetuning 有点类似

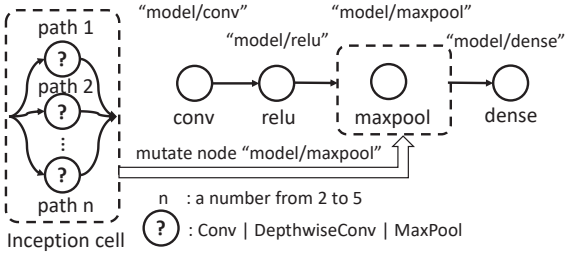


Figure 4: Base model and mutation: An example.

```
create_node(name:str,op:Op,params:dict={})
delete_node(node:Node)
connect(src:NodeOutput,dst:NodeInput)
del_connect(src:NodeOutput,dst:NodeInput)
update_node(node:Node,op:Op=None,params:dict={},
            inputs:list=None)
choose(candidates:list,n_chosen:int=1,
       type:str="choice",ctx:dict=None)
```

Figure 5: Mutation primitives and the `choose` primitive.

a jumbo model, it also becomes challenging to scale the training of this jumbo job to multiple GPUs and servers [27, 50]; in contrast, scaling to multiple GPUs is straightforward for a set of individual jobs, each training a single model.

Insufficiency of existing AutoML systems. Previous AutoML systems (*e.g.,* Google Vizier [24], Auto-WEKA [61], Auto-Sklearn [22]) abstract the AutoML problem as hyper-parameter tuning. Although a certain NAS problem can be modeled as the tuning of specific hyper-parameters, it often involves the definition of an ad-hoc set of hyper-parameters, making it cumbersome to express different model spaces in a general way. It is especially painful to hyper-parameterize evolutionary NAS [13, 23, 51] where neural architectures can randomly evolve. Moreover, the expressed architectures are hardly understood by compilers, making optimizations almost impossible. Some recently emerged AutoML systems (*e.g.,* AutoKeras [32]) provide more support to NAS. They can automatically search neural architectures but with specifically implemented model spaces and exploration strategies, where system optimizations are hardly applicable.

Retiarii is designed to address the abovementioned pains. It provides great expressiveness to support various model spaces and strategies in a systematic and programmable way. It clearly decouples model space from exploration strategy and enables system optimizations to speed up exploration process.

3 Mutator as the Core Abstraction

Exploratory-training is all about exploring a model space. Mutator is the core abstraction that connects the specification and exploration of a model space, while exposing the correlations between models for further optimizations.

Base models. Retiarii follows the standard practice of characterizing a DNN model as a data-flow graph (DFG), where each node represents an operator (or a subgraph) with one or multiple input and output tensors and an edge connects an output tensor of a node to an input tensor of another node.

Retiarii introduces the notion of *base models* as the starting points of an exploratory-training and preserves the way a single DNN model is specified for base models. In fact, Retiarii can simply import base models defined in an existing deep learning framework such as TensorFlow. Figure 4 illustrates an example base model with a chain of 4 operators (a convolutional neural network).

Mutator. Exploratory-training is typically a process of applying modifications (*e.g.,* as depicted in Figure 1) to existing models, starting from base models. Rather than encoding modifications in a complex jumbo model, Retiarii cleanly separates modifications from the original target models and encode each as a Mutator, an abstraction designed to be expressive, modular, reusable, and composable. The model space to be explored by an exploratory-training process is then the base models, plus all the resulting models from applying mutators to the base models and to any subsequently generated models.

Graph matching and manipulation in Mutator. Each mutator specifies matching criteria to identify subgraphs of a target model's DFG to operate on, followed by a series of graph construction operations to modify the matched subgraphs to create a new model. The mutator abstraction can also use the `choose` primitive to describe different options to choose from in a mutator, so that the mutator can produce a number of models without duplicating the mutator code to create a new mutator for each option.

Retiarii's current graph matching is based on node type or node name, which is simple, but sufficient for all the use cases we have implemented. But it can be extended easily to more expressive graph matching if necessary.

Retiarii introduces general mutation primitives like `create_node` for a mutator to manipulate the node and edge in a model. The primitives are summarized in Figure 5. Note that a node in Retiarii can also represent a subgraph. Thus the primitives can also be applied to a subgraph (*e.g.,* a layer or a cell) of a model.

For each model instantiation, Retiarii records all the mutation primitives called in a mutator. Hence Retiarii can easily identify model correlations across instantiated models. For example, between two instantiations of the same base model, the nodes *not* modified by the mutator are considered identical. Retiarii can leverage such information to optimize the multi-model training (details in §5).

Mutator: an example. Figure 4 depicts a model space in which the third node ("model/maxpool") of the base model

```
1  # define the graph mutation behavior
2  class InceptionMutator(BaseMutator):
3    def __init__(self, paths_range, candidate_ops):
4      self.paths_range = paths_range # [2, 3, 4, 5]
5      self.ops = candidate_ops # {conv, dconv, ...}
6    def mutate(self, targets):
7      if not three_node_chain(targets):
8        return err
9      n = choose(candidates=self.paths_range)
10     delete_node(targets[1])
11     for i in range(n): # create n paths
12       op = choose(candidates=self.ops)
13       nd = create_node(name='way_'+str(i), op=op)
14       connect(src=targets[0].output, dst=nd.input)
15       connect(src=nd.output, dst=targets[2].input)
16
17 # mutation applied to the graph
18 apply_mutator(targets=["model/relu", "model/
       maxpool", "model/dense"],
19     mutator=InceptionMutator(
20     [2, 3, 4, 5], [conv, dconv, pool]))
```

Figure 6: A mutator that constructs an Inception-like cell.

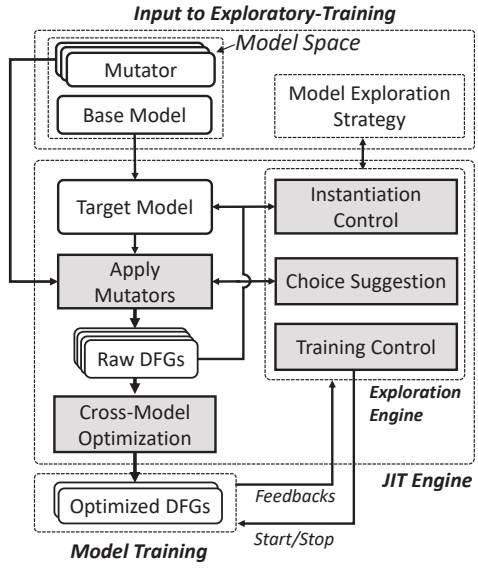


Figure 7: The architecture of Retiarii.

can be mutated with a multi-path cell. The cell could have 2 to 5 paths, each of which chooses an operator from *Conv*, *DepthwiseConv* and *Maxpool*. Figure 6 shows the code of the mutator, *i.e.,* `InceptionMutator`, which implements the model space illustrated in Figure 4.

All the mutation logic is encapsulated in the `mutate` function (lines 6-15). The entry point of the mutator is given by `targets` in the `mutate` function (line 6 of Figure 6), to match nodes/subgraphs in the given model. The `targets` of `InceptionMutator` is a chain of 3 nodes. This shows that a mutator can be applied to a subgraph with a specific pattern, which improves the reusability of a mutator. In the example code, the mutator first ensures that the matching is a chain of 3 nodes (lines 7-8). It will then call `choose` (line 9) to select an integer n to create n paths subsequently. On creating each path, the mutator will call `choose` (line 12) again to select an operator for the node in the path. Note that the code for a mutator can contain arbitrary complex control flow in a mutator (*e.g.,* the control loop in lines 11-15 of Figure 6), without polluting the instantiated models, unlike in the case of jumbo models with control flows. Finally, a call to `apply_mutator` will create a mutator instance (line 18), which matches a chain of *relu*, *maxpool*, and *dense*.

4 Retiarii Just-In-Time Engine

A key design decision for Retiarii to support exploratory-training is to instantiate models to explore on the fly and manage the training of instantiated models dynamically. This is accomplished by Retiarii's just-in-time (JIT) engine (Figure 7), which takes as input one or more base models, a set of mutators, and a policy describing the exploration strategy.

The end-to-end exploratory-training process is driven by the policy described as a *Model Exploration Strategy*. The JIT engine maintains a set of *target models*, initialized with the set of base models, and consults the model exploration strategy to decide which target model(s) and mutator(s) to choose (*i.e., Instantiation Control*), as well as which choices to make for each `choose` within those mutators (*i.e., Choice Suggestion*), to instantiate new models. The decision can be guided by a context-free strategy (*e.g.,* making a random choice upon each `choose`) or by a history-based strategy, generating choices based on which models have been previously instantiated [60]. The `choose` interface in Mutator enables the customization of the choices.

Once new models are instantiated (*i.e., Apply Mutators*) as Raw DFGs, the JIT engine transparently performs *Cross-Model Optimization* (§5). Because the JIT engine records the mutation history, the Cross-Model Optimization module can easily detect identical nodes across models to produce optimized DFGs by applying common sub-expression elimination [44], cross-model operator batching [15, 41], and NAS optimizations (§5). The optimized DFGs are then converted to the standard model format for the existing deep learning frameworks to perform single-model optimizations before training. In *Training Control*, the JIT engine launches training on new models, monitors the training of instantiated and optimized models, collects training feedbacks (*e.g.,* model accuracy), adjusts training priorities and resource allocation, and terminates training of less promising models, all guided by a model exploration strategy.

Retiarii's Mutator abstraction and JIT engine offers an elegant architecture to support exploratory-training, following the principles of separating policy from mechanisms and separation of concerns, and maximizing modularity, reusabil-

ity, and opportunities for optimizations. In addition to the common functionalities (*e.g.,* Cross-Model Optimization) in the overall infrastructure, mutators and policies encoding the model exploration strategies might also be reused. This is in sharp contrast to the current practice of encoding everything in a jumbo code, which is hardly understandable or reusable due to tight coupling.

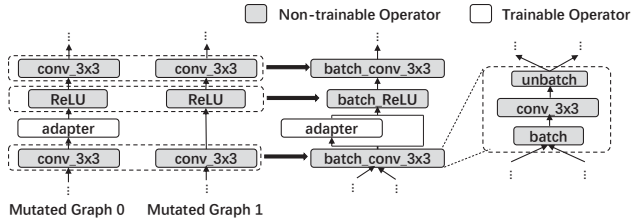


Figure 8: Operator batching: An example.

或许可以设计两个继承 torch.autograd.Function 的 methods，在 forward时将 tensor 沿 batch dim 拼接/拆分，在 backward 时对 tensors 分别计算梯度传输?

5 Cross-Model Optimization

The DNN models instantiated by Retiarii in an exploratory-training process tend to have significant similarities as their DFGs share common subgraphs, thereby offering huge opportunities for Retiarii to optimize the training of multiple models. With mutators that identify and record all modifications to a model's DFG, Retiarii can easily find the common subgraphs of multiple DFGs, circumventing the generally NP-hard and APX-hard problem of identifying maximum common subgraphs [34].

5.1 Cross-Model Optimization Opportunities

Three different cross-model optimization opportunities are identified, depending on the inputs, weights, and trainability[2] of operators in the common subgraphs.

Common Sub-expression Elimination (CSE). CSE is a common compiler optimization to eliminate identical operations of a program by only computing them once. CSE can be applied to the non-trainable operators in the common subgraphs with common inputs and outputs, but cannot be applied to trainable operators as their weights will change during training, rendering their computation different after the first iteration. In practice, we find CSE particularly useful for merging prefix nodes of a DFG, because they are often non-trainable operators for data loading and preprocessing, as neural architecture search often uses the same dataset, batch size, and preprocessing procedures. When running multiple data-flow graphs concurrently on a single server, CSE can also avoid contention on shared storage and CPUs to maximize utilization of expensive GPUs.

或许可以用在 multi-tasks 下没有 adapter hooked 的 modules 上，每个 task 需要独立执行这些 frozen modules.

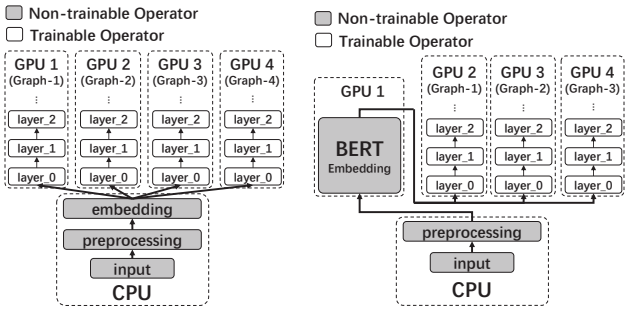
Operator Batching. Common operators with different inputs and weights can potentially be batched together and computed in a single operator kernel. This optimization is useful for model exploration in multi-domain deep learning and transfer learning [28, 52, 53]. In this scenario, a model is modified to a new task with only minor changes, thus those modified models usually share a common skeleton. Adapter-based transfer learning is a one such example: networks have the same architecture from a pre-trained network, with adapters inserted at different locations. Only the inserted layers are fine-tuned [28, 52, 53]. Figure 8 illustrates an example that two graphs share multiple layers with the same weights. After merging the two graphs, the input of the common operators are batched along the batch dimension, and the output of the batched operators are split before adapters. Common operations with different weights (*e.g.,* trainable weights) can also be batched by leveraging special kernels (*e.g.,* grouped convolution [37], *batch_matmul*) that can parallelly compute on slices of an input tensor. This allows Retiarii to enable more fine-grained sharing of GPUs by increasing SIMD utilization with less GPU memory.

Super-Graph for Weight Sharing. Weight sharing is a machine-learning optimization shown to deliver improved empirical performance for certain model training: instead of training a graph's weights from scratch, shared weights are inherited from other graphs to continue the training in this graph [27, 50]. Retiarii naturally supports this training optimization by allowing model developers to annotate operator weights they want to share. Retiarii will automatically identify the weight sharing-enabled operators in common subgraphs. The DFGs with shared weights will be merged to build a super-graph. By training the super-graph together in one DFG, Retiarii can avoid overhead of checkpointing shared weights, because with weight sharing each graph has short training time (*e.g.,* several mini-batches). To accelerate the training of the merged super-graph, we further introduce a new type of parallelism when constructing executable graphs (§5.2) by increasing its scalability on distributed GPU clusters. Note that super-graphs are generated and used for optimizations only, and not exposed to developers.

5.2 Executable Graph Construction

To exploit these cross-model opportunities, Retiarii needs to construct executable graphs from the raw DFGs. The construction involves decisions of model merging, device placement of operators, and training parallelism, constrained by physical environment (*e.g.,* server configuration). Retiarii adopts a policy similar to Gandiva [64] that introspectively selects graphs to merge. Moreover, Retiarii specifically optimizes device placement of CSE-optimized graphs and training parallelism

[2] Similar to most popular deep learning frameworks, Retiarii allows model developers to specify whether the weights associated with an operator are *trainable*, whose weights will be applied with gradients during back-propagation.



(a) CPU-based embedding (b) GPU-based embedding

Figure 9: Device placement for CPU/GPU-based pre-trained embedding when constructing executable graphs.

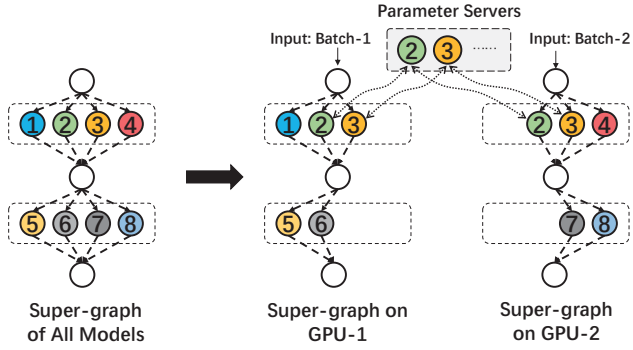


Figure 10: Retiarii uses mixed parallelism to improve scalability of weight sharing-based training.

of weight sharing.

Device Placement of CSE-Optimized Graphs For DFGs sharing the same dataset and preprocessing, these common operators can be merged by common sub-expression elimination. The most efficient execution plan of merged graphs depends on the types of merged operators and configuration of GPU servers. Figure 9 shows two different execution plans of CSE-optimized graphs. Both examples use a pre-trained embedding before the trainable layers. The difference is that the embedding in 9(a) is CPU-based (*e.g.,* word2vec [43]) while the embedding in 9(b) is GPU-based (*e.g.,* BERT [10, 19]). When BERT-embedding is the bottleneck of model computation and consumes most of GPU memory, dedicating one GPU for it can improve the pipeline and reduce memory consumption. Thus, we may pack more graphs on the rest of GPUs to improve the training throughput. Retiarii currently uses a whitelist to identify operators that require dedicated GPUs. We leave the automatic graph partitioning and optimization to future work.

In Retiarii, all cross-graph optimizations are applied within every batch of models. We first profiled the iteration time, peak GPU memory, and GPU utilization of each model by independently running for a few iterations. Then the models are sorted based on the iteration time. Retiarii greedily packs as many models as possible into one GPU. If the excutable graph's total training throughput is lower than that before optimization, the optimization will be reverted.

Mixed Parallelism for Weight Sharing. Weight sharing suffers from the scalability issue. After an exploration strategy instantiates a set of models, these models need to be trained sequentially (in an interleaved way) with different data to guarantee that every model can use the latest version of shared weights without losing training accuracy. A single model can hardly scale to a large number of GPUs using data parallelism, because a large batch size would harm model accuracy [25, 35]. Figure 10 shows an example of how Retiarii trains weight-shared models on two GPUs. Retiarii can improve the scalability by splitting the super-graph onto multiple GPUs, when the super-graph of all models is too large to fit into one GPU. Retiarii spreads the instantiated models into multiple super-graphs (each on a GPU) to be trained together. This can be regarded as model parallel training of the super-graph of all models. Moreover, in each iteration, models in different GPUs will be fed with different batches of data (like data parallelism), following the requirement of weight-shared training. The shared weights will be synchronously updated using parameter servers by averaging their gradients. Note that, it is difficult to apply Retiarii's mixed parallelism to a jumbo model, since a compiler can hardly understand and partition the sophisticated jumbo model without knowing each individual model's architecture. Our evaluation in §7.4 shows Retiarii's mixed parallelism yields better scalability that reduces the training time by up to 8.58× without affecting validation accuracy, compared with the traditional approach that trains the jumbo model using data parallelism.

6 Implementation

We have implemented Retiarii in about 19,723 lines of code, in which about 5,436 lines of code for the core Retiarii JIT engine, 5,203 lines of code for model, state, data management with failure recovery, and 9,084 for managing training with interfaces to various training services, such as Kubeflow [2]. We also wrote an additional 6,157 lines of code to implement 11 exploration strategies, 6 mutator classes, and 27 model spaces [4].

Building internal representations of base models and mutators. Our implementation supports base models defined in PyTorch and TensorFlow, which we convert to their graph representations. The conversion is done through TorchScript [9] for PyTorch. TensorFlow naturally supports a similar graph representation and offers the utility to output in a protobuf format. We do not yet support dynamic graphs. The mutators are extracted through Python Abstract Syntax Trees (AST) [1].

```
class ExplorationStrategy:
  # the APIs for instantiation control
  def generate_graph(self, new_graph_id)
  def on_ask_target_graph(self, graph_id)
  def on_ask_choice(self,graph_id,type,values,ctx)
  # the APIs for training control
  def execute_graph(self, graph_id, load_ckpt)
  def terminate_graph(self,graph_id,do_ckpt)
  def on_ask_training_approach(self, graph_id)
  # the APIs for getting provisioned information
  def query_mutation_history(self, graph_id)
  def on_receive_feedbacks(self,graph_id,feedback)
```

Figure 11: Some key APIs for an exploration strategy.

The base graph and mutators are then passed to the JIT engine.

Materializing the JIT engine. The JIT Engine drives the whole exploratory-training process. It first starts an exploration strategy which is an independent executable Python script. The strategy uses the APIs listed in Figure 11 to interact with the engine. Users are free to customize a new one following the interface. For instantiating a model, the mutators are applied one after another. On applying a mutator, the JIT engine retrieves the subgraphs specified by `targets`, and feeds them into the mutator. The instantiation is guided by an exploration strategy through those callback functions (*i.e.,* "on_*"). The JIT Engine maintains all the instantiated and trained models in a data store (*i.e.,* SQLite in our implementation) and collects runtime information of those models, such as model accuracy, execution time, which can by queried by the exploration strategy. Each model can have its training approach, *e.g.,* a training loop with a configured epoch number and batch size. We follow the practice in PyTorch Lightning [8] to provide a wrapper for programming and configuring a training approach. An exploration strategy can specify the training approach for each instantiated model.

Converting models for training. In our implementation, the optimized graphs are trained on current deep learning frameworks, such as TensorFlow and PyTorch. To make the optimized graph executable on those frameworks, we implement a converter to translate an optimized graph into TensorFlow or PyTorch code. Taking PyTorch as an example, the optimized graph is converted to a PyTorch module, *i.e.,* graph nodes in `__init__()` and connections in `forward()`. In cases where an optimized graph could contain multiple models, the losses are either added or concatenated to produce a single one. We enable device placement for a model with each framework's utility, such as the `to()` method in PyTorch and `with tf.device()` in TensorFlow.

Distributing exploratory-training. Exploratory-training usually requires lots of computation resources. In our implementation, Retiarii's JIT engine runs on a single machine, while the instantiated models can be distributed to wherever computing resources are available (*e.g.,* a cluster). For training of each model, Retiarii implements a wrapper to monitor its execution and collects metrics (*e.g.,* training performance) to report back to the JIT engine.

Tolerating and handling failures. As exploratory-training is usually time-consuming, in our implementation we deal with failures of both the JIT engine and model execution. All the exploration history is kept in the data store. When the JIT engine fails, it will be restarted and recover the state of exploration strategy by replaying the data in data store. For model exploration, the most valuable data are the set of models that have been explored and their observed results. These data are usually enough to continue an interrupted exploration from a previously known state. For an exploration strategy that maintains its own, additional states that cannot be recovered by our automatic mechanism, its own recovery logic must be provided. Another type of failure comes from the optimized graphs. If the execution of an optimized graph fails (*e.g.,* out of GPU memory, tensor shape mismatch), while each model in this graph runs without error, Retiarii will revert to running the individual models separately.

Limitations. Retiarii has limited support to dynamic graphs. Retiarii's mutators are applied to a base model. However, sometimes it is difficult to extract a graph representation from the a highly dynamic base model (*e.g.,* Tree-LSTMs [58]). Also, the current implementation of operator batching is limited. Some operator batching is possible but is not implemented as it requires implementing new GPU kernels. Moreover, when a model is mutated, it requires additional programming efforts to match the shape of adjacent layers' input/output tensors. It is currently out of the scope of Retiarii to handle possible shape mismatch after mutation. We leave automatic shape inference and matching to our future work.

7 Evaluation

We evaluate the performance of Retiarii for exploring neural network architectures. Overall, the key findings include:

- The separation of model space and exploration strategy makes it easy for Retiarii to try different combinations. Retiarii currently supports 27 popular Neural Architecture Search (NAS) solutions. Most of them can be implemented by the three mutator classes provided by Retiarii.
- A number of micro-benchmarks show how Retiarii's cross-model optimizations greatly improve training efficiency.
- Retiarii improves the model exploration speed of three NAS solutions by up to 2.58×, compared with traditional approaches.
- Retiarii improves the scalability of weight sharing-based NAS solutions and brings up to 8.58× speed-up using the proposed mixed parallelism, compared with data parallelism.

NAS Solution	Model Space	Exploration Strategy	Required Mutator Class			
			Input Mutator	*Operator Mutator*	*Inserting Mutator*	*Customized Mutator*
MnasNet [59]	MobileNetV2-based space	Reinforcement Learning		✓	✓	
NASNet [70]	NASNet cell	Reinforcement Learning	✓	✓		
ENAS-CNN [50]	NASNet cell variant	Reinforcement Learning	✓	✓		
AmoebaNet [51]	NASNet cell	Evolutionary	✓	✓		
Single-Path One Shot (SPOS) [27]	ShuffleNetV2-based space	Evolutionary		✓		
Weight Agnostic Networks [23]	Evolving space w/ adding/altering nodes adding connections	Evolutionary		✓		✓
Path-level NAS [13]	Evolving space w/ replication and split	Reinforcement Learning				✓
TextNAS [62]	TextNAS space	Reinforcement Learning	✓	✓		
...	...	...	...	...	...	...

Table 1: NAS solutions currently supported by Retiarii, and required mutators to implement them in Retiarii. Please refer to [4] for the full list that contains 27 NAS solutions in total.

7.1 Expressiveness and Reusability

Table 1 shows 8 out of 27 NAS solutions currently supported by Retiarii (please refer to [4] for the full list). After decoupling model spaces from exploration strategies, developers can easily reuse them without extra coding efforts. For example, the exploration strategy "reinforcement learning" is reused by MnasNet [59], NASNet [70] and ENAS-CNN [50]. Several machine learning researchers at Microsoft Research are now using Retiarii to explore more NAS solutions by leveraging different combinations of model spaces and exploration strategies.

To build these model spaces, Retiarii provides three default mutator classes. *Input Mutator* is to mutate inputs of matched operators. *Operator Mutator* is to replace matched operators with other operators. *Inserting Mutator* is to insert new operators or sub-graphs after matched operators. We find the three mutator classes are enough to implement most of the listed NAS solutions. Moreover, Retiarii allows model developers to build customized mutator classes using basic graph mutation primitives to implement more complex model spaces, *e.g.,* Weight Agnostic Networks [23] and Path-level NAS [13].

7.2 Micro-benchmarks

7.2.1 Shared Data Loading and Preprocessing

The following experiments demonstrate two micro-benchmarks of common sub-expression elimination, where multiple models share the same data loading and preprocessing. These micro-benchmarks are evaluated on 4 V100 GPUs with 20 CPU cores. We compare Retiarii with a baseline that runs each model independently without CSE. For a fair comparision, CUDA Multi-Process Service (MPS) [5] is enabled for the baseline when Retiarii decides to packed more than one model in a GPU.

Avoiding CPU Bottleneck. Figure 12 shows the aggregate throughput and CPU usage with the increased number of MnasNet0.5 (*i.e.,* depth multiplier=0.5) models [59] running concurrently on the 4 V100 GPUs and 20 CPU cores. The models are trained on ImageNet with a batch size of 224 with the same preprocessing as in [59]. The baseline approach runs each model independently, thus each batch of data will be loaded and preprocessed multiple times. Retiarii merges the data loading and preprocessing across different models thus they are executed only once. Both Retiarii and the baseline can further pack multiple models into one GPU to run them concurrently. The models are distributed in a round-robin way. For example, when running 6 models, the first two GPUs are packed with two models on each GPU, while each of rest two GPUs runs only one model. The measured performance is averaged over one training epoch.

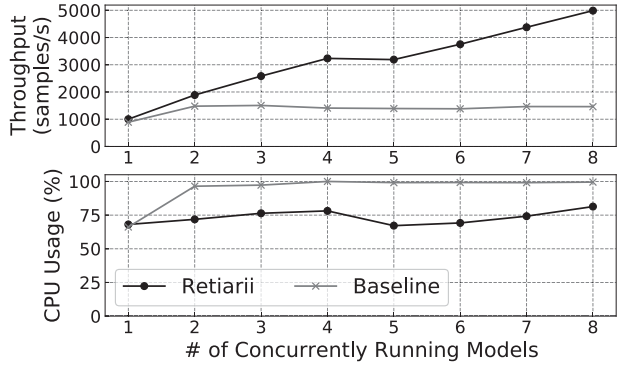


Figure 12: The aggregate throughput and CPU usage with varying number of concurrently running MnasNet0.5 models.

Overall, Retiarii increases the throughput by 3.41× when running 8 models on 4 GPUs. The bottom figure in Figure 12 shows that training one MnasNet0.5 model has already consumed about 75% of CPU cores. Thus, CPU will become

the bottleneck when running more than one model without sharing. On the contrary, Retiarii eliminates the redundant data loading and preprocessing. Increasing the number of concurrent models does not affect the CPU usage for data loader. The marginal increase of CPU usage in Retiarii is due to other computations, which can not be merged (*e.g.,* overhead of the training runtime).

Also note that, running 5 models does not bring higher throughput than running 4 models. This is due to the overhead of synchronization brought by unbalanced model assignment, *i.e.,* the first GPU is packed with two models while each of the rest three GPUs runs only one model. In Retiarii, merging the graphs will force them to be trained synchronously. Packing two models in one GPU may increase their iteration time, thus the rest three GPUs have to wait for the two slower models in the first GPU in every iteration. This suggests to merge the graphs with a similar iteration time to avoid severe synchronization overheads.

Avoiding GPU Bottleneck. Non-trainable embedding [49] can be regarded as a special type of data preprocessing. In this micro-benchmark, we use BERT [19] to obtain pre-trained contextual embeddings of input tokens from Stanford Sentiment Treebank (SST) dataset [55] for training TextNAS [62], which is one of the state-of-the-art natural language processing models. The batch size for each TextNAS model is 128. Different from the micro-benchmark of avoiding CPU bottleneck, the embedding computation is placed on GPU because the BERT embedding runs much faster on GPU than CPU [3]. The baseline still runs multiple models independently. The performance is measured by averaging over one training epoch.

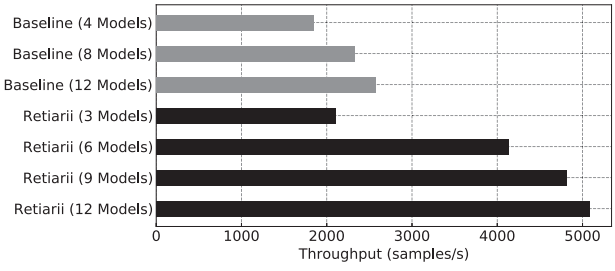


Figure 13: The aggregate throughput with varying number of TextNAS models.

Figure 13 shows the result. Overall, Retiarii achieves 1.97× throughput of the baseline when training 12 TextNAS models on 4 V100 GPUs. Both the baseline and Retiarii meet out-of-memory when running more than 12 TextNAS models. As we have shown in Figure 9, Retiarii uses model parallelism to dedicate one GPU to compute the BERT embedding, which is pipelined with the training of TextNAS models on the other three GPUs. Since the BERT embedding is the bottleneck in each training iteration, this optimization allows the training of more TextNAS models to be overlapped with the BERT

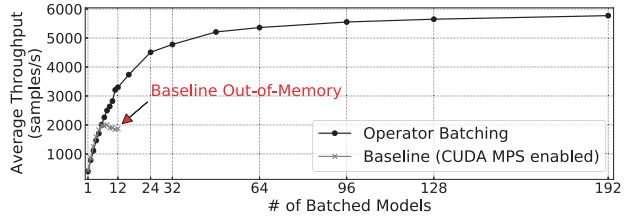


Figure 14: The aggregate throughput with varying number of batched models.

embedding. In this experiment, we find Retiarii can pack two TextNAS models on each GPU (*i.e.,* 6 models in total) without affecting the iteration time. And 12 models can be packed in total with better aggregated throughput, but each model's iteration time is degraded. Although the baseline can also pack up to 12 models on 4 GPUs, the compute-intensive BERT embedding repeats three times per GPU that greatly increases the iteration time. Only marginal improvement on throughput is observed in the baseline when packing more models using CUDA MPS.

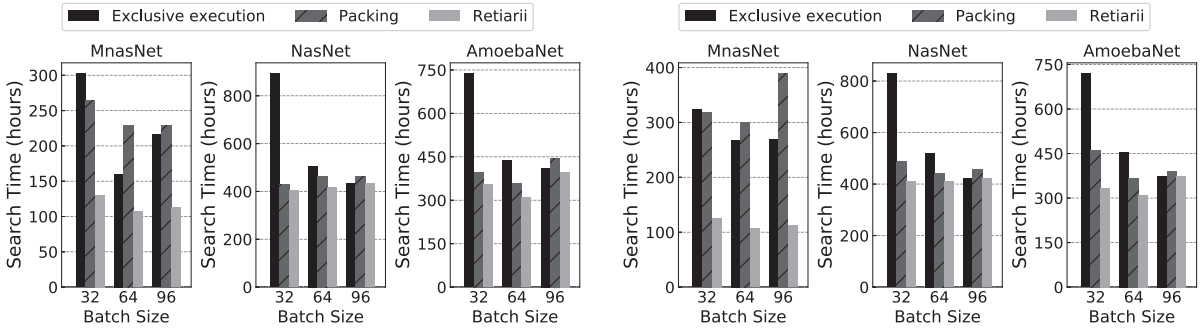
7.2.2 Operator Batching

To evaluate operator batching across graphs, we insert adapter layers to a pre-trained MobileNet [29]. Weights from the MobileNet are fixed during training. These models use the same batch size, which is 8 images per mini-batch. Synthetic data without preprocessing is used to avoid the gain from shared data loading. The models are trained on one V100 GPU of 16GB GPU memory. Similar to previous micro-benchmarks, the baseline uses CUDA MPS to execute multiple models in one GPU. The performance is measured by averaging the throughput over 1500 mini-batches.

Figure 14 shows the average throughput of concurrently running models. Overall, Retiarii's operator batching improves the aggregate throughput by 3.08× when batching 192 models, compared with the baseline that can only train at most 12 models together. Retiarii can batch more models than the baseline because it only has one copy of (fixed) weights from MobileNet. Only the memory for adapters is increased when batching more models. Even when Retiarii batches 12 models, it still achieves 1.76× improvement on the aggregate throughput. This improvement comes from the benefit of vectorization to execute the batched operators in a single kernel, which increases GPU utilization.

7.2.3 Optimization for Weight Sharing

To evaluate Retiarii's optimization for weight sharing, we use Single Path One-Shot (SPOS) [27] to explore a model space built by ShuffleNetV2 blocks, where a model is instantiated for every batch of data. The models are trained with



(a) NVIDIA Data Loading Library (DALI)

(b) PyTorch DataLoader

Figure 16: The completion time of the search phase of three NAS approaches, each of which generates 1,000 models for training.

synthetic data on a V100 GPU of 16GB GPU memory. We implemented two baselines that share weights of overlapped operators among the instantiated models through weight saving and loading. In the first baseline, a checkpoint file is used for weight sharing, *i.e.,* a model loads its weights from the file, then saves its updated weights to the file after training a mini-batch. In the second baseline, the file is replaced with a `dict` object located in GPU memory. Both model weights and optimizer states (*e.g.,* momentum) need to be checkpointed.

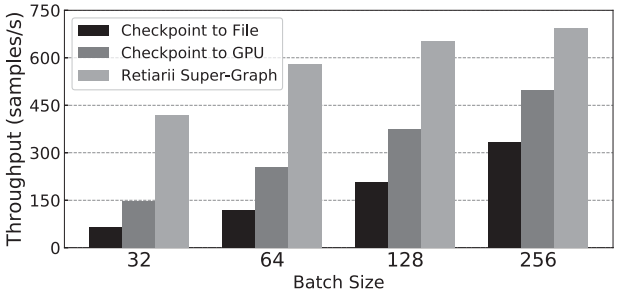


Figure 15: The throughput of weight sharing with and without cross-model optimization.

The result is shown in Figure 15. By merging multiple models as a super-graph, Retiarii's cross-model optimization improves the throughput by up to 6.52× when batch size is 32, and 2.08× when batch size is 256 (compared with checkpoint-to-file). Since SPOS only trains an instantiated model with a batch of data, frequent checkpointing brings significant overheads. Merging instantiated models into a super-graph allows Retiarii to load the models only once (at the beginning). Thus, Retiarii can use control flow to only activate the desired model, which also saves the time of model initialization. The performance of a jumbo model is similar to that of Retiarii's super-graph, the difference is that the super-graph is automatically built by Retiarii's JIT engine while the same graph is manually programmed in the jumbo model approach. This leads to a big performance gap on parallel training which will be illustrated in §7.4, as Retiarii fully understands each sampled graph and the weight sharing pattern.

7.3 Speeding up Neural Architecture Search

To evaluate the performance of running NAS solutions on Retiarii, we select three popular and representative NAS solutions from Table 1: (1) MnasNet [59], (2) NASNet [70], and (3) AmoebaNet [51]. They cover different combinations of model spaces and exploration strategies: MnasNet and NASNet use the same search strategy; NASNet and AmoebaNet have the same model space. We compare Retiarii against the one-off solutions built by traditional deep learning frameworks. Since Retiarii separates the cross-model optimization from model generation, all the three NAS solutions can leverage the three cross-model optimizations in §5 to accelerate the search of architectures without extra effort.

To evaluate the traditional approaches, which are unaware of cross-model relations, we test the following two baselines. (1) **Exclusive execution**: a model is trained independently and exclusively on one GPU. (2) **Packing**: multiple models may share the same GPU using CUDA MPS without merging their graphs; it uses Retiarii's decisions to choose the models to be packed onto the same GPU.

In the experiments, each NAS approach will generate 1,000 models in 20 batches (each batch contains 50 models). For a fair comparision, Retiarii and the two baselines are given the same set of models in the same order. We compare the time to finish the training of all the 1,000 models for 1 epoch on ImageNet's training images [18]. We vary the batch size from 32 to 96 (batch size larger than 96 will lead to Out-of-Memory). We also compare the performance using two implementations of the data loader, *i.e.,* NVIDIA Data Loading Library (DALI) [6] and PyTorch DataLoader [7], to understand the impact of data loading. The experiments are conducted on 4

NVIDIA Tesla V100 GPUs (each has 16GB GPU memory). Simlar to [64], we "fast-forward" the experiment by instructing model trainings to skip a number of iterations when the iteration time becomes stable. We measure the average iteration time over 100 mini-batches, which is used to project the entire training time.

Figure 16 shows the results of the search time of different settings. In most settings, Retiarii is substantially faster than the two baselines due to the cross-model optimizations. Overall, Retiarii achieves up to 2.57× speed-up on the search time compared with the two baselines. The performance gain mainly comes from packing multiple models in the same GPU, and shared dataloading and preprocessing. Because the packing baseline is agnostic to the cross-model relations, it cannot apply cross-model optimizations thus only brings up to 1.42× speed-up over the exclusive execution. Moreover, due to the increased CPU burden on the larger batch size, the packing baseline runs even slower by 31% than the exclusive execution on PyTorch DataLoader when the batch size is 96. Note that, an introspective policy, *e.g.,* Gandiva [64], can remedy the packing baseline's slow-down by reverting the packing when the training speed is slower. But the key insight in this experiment is that only using packing without cross-model merging will limit the space for improvement.

Retiarii achieves higher speed-up on MnasNet than NASNet and AmoebaNet. Because the models from MnasNet are designed for mobile devices that have a lower GPU memory usage and shorter iteration time, Retiarii can pack more MnasNet models into one GPU and merge their graphs for cross-model optimizations. As the generated models have different memory consumption, the number of models that can be fit in the same GPU varies accordingly. When the batch size is 32, Retiarii can run 4-22 MnasNet models simutanously; but only 4-8 NASNet/AmoebaNet models due to the larger model size. We also observe Retiarii achieves higher speed-up on PyTorch DataLoader, because DALI is more efficient on data preprocessing that reduces the probability of bottleneck on CPU.

7.4 Scaling Weight-Shared Training

In addition to system optimizations, Retiarii also enables and enhances the weight sharing optimization advocated by the machine learning community. As shown in §7.2.3, Retiarii builds a super-graph for weight sharing to avoid the overhead of model building and checkpointing. This optimization can be further improved by training the super-graph using mixed parallelism to scale it to a GPU cluster.

In this experiment, we build a model space with ShuffleNetV2 blocks described in the Single Path One-Shot (SPOS) paper [27]. Each model in the model space is randomly sampled and trained for one batch of data [17, 27]. The models are trained for 60 epochs in total on the ImageNet dataset (with 1,281,167 images). As a result, a new model is instantiated for every batch of data (*e.g.,* $1281167/256 \times 60 = 300240$ models are instantiated when the batch size is 256). The experiment runs on two servers, each has 4 V100 GPUs. We use the common evaluation metric of weight sharing-based approaches [12, 27] to evaluate the searched space. We randomly sample 196 models and evaluate each model using 256 images from ImageNet's validation set. Then we calculate the average validation accuracy of the 196 models. The higher the average validation accuracy is, the better the space is explored. We compare Retiarii's mixed parallelism with three commonly used data parallelism approaches. To understand the benefit of mixed parallelism, all the three baselines of data parallelism and Retiarii's mixed parallelism enable the super-graph optimization (*i.e.,* no saving and loading of weights). Specifically, the former three are manually programmed jumbo-models, while the latter is a super-graph automatically built by Retiarii.

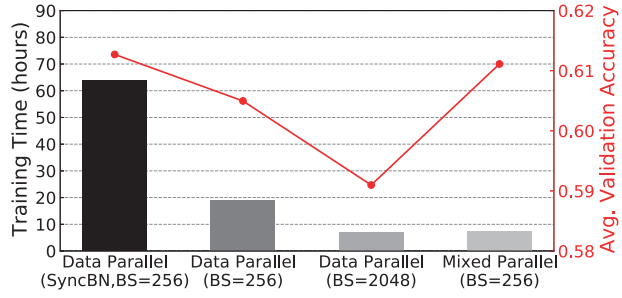


Figure 17: Training time and validation accuracy of weight sharing. The left y-axis shows the training time (bar chart). The right y-axis shows the validation accuracy (line chart).

Figure 17 shows the training time and validation accuracy of the three data parallelism approaches and Retiarii's mixed parallelism. The data parallelism of the left two bars and Retiarii's mixed parallelism use the batch size of 256 with a learning rate of 0.125 per model (or per 256 data samples). As a common practice of data parallelism, scaling to 8 GPUs requires to split each batch of data across the 8 GPUs (*i.e.,* the batch size per GPU is 32). SyncBN [66] is an optimization to calculate batch normalization across multiple GPUs, which proves to improve the model quality, but slows down the training due to intensive synchronization and data transmission across GPUs. As shown in Figure 17, SyncBN-based data parallelism requires more than 60 hours of training time. Disabling SyncBN reduces the training time to $\sim$ 20 hours but harms the model accuracy. In contrast, Retiarii's mixed parallelism greatly reduces training time (only 7.45 hours), achieving up to 8.58× speed-up over SyncBN-based Data Parallel training. This is because the mixed parallel training avoids the synchronization overhead of SyncBN as each GPU runs a different model requiring no cross-GPU synchroniza-

tion. Moreover, Retiarii's mixed parallel training produces a comparable validation accuracy to SyncBN-based Data Parallel training (61.49% v.s. 61.11%). Another practice of data parallelism is to increase batch size and learning rate with the increased number of GPUs. The result is shown as the second bar on the right of Figure 17. Although the training time is reduces to 7.04 hours, the model's validation accuracy degrades significantly. This result is consistent with the common wisdom in deep learning community that larger batch size could hurt model accuracy [25, 35]. In summary, Retiarii's mixed parallelism achieves better scalability for weight-shared training, without sacrificing model accuracy.

8 Related Work

Deep Learning Frameworks. Deep learning frameworks (*e.g.,* PyTorch [48], TensorFlow [11], and MXNet [14]) are designed to describe and train an individual DNN model, which covers only one step in the end-to-end exploration-training process of devising a high-quality model.

Network Architecture Search Algorithms. To automate the design of neural networks, Neural Architecture Search (NAS) [39, 50, 59, 60, 69, 70] develops algorithms to discover the state-of-the-art neural model architecture. Limited by the existing deep learning frameworks, their implementations often couple model space, exploration strategy, and model training together, introducing barriers to innovations and optimizations. In contrast, Retiarii's modular and decoupled approach maximizes reusability and facilitates optimizations.

AutoML Systems. Automated Machine Learning (AutoML) automates the end-to-end process of real-world machine learning problems, *e.g.,* AutoGluon [21], TPOT [47], Auto-Sklearn [22], Auto-WEKA [61], AutoKeras [32]. The implementations of these systems still couple the domain-specific model space and exploration strategy, making it hardly reusable to other problem domains.

The hyper-parameter tuning systems like Google Vizier [24] and Katib [68] can be used for neural architecture search. To use a hyper-parameter tuning system, the model space and exploration strategy are being parameterized. Since different model space and exploration strategy often use a different set of parameters, this approach limits the reusability of the implementation. Moreover, the hyper-parameter tuning approach can limit the expressiveness of the system as well. Some model space is hard to be parameterized, *e.g.,* evolutionary NAS [13,23,51]. It is worth noting that Retiarii's Mutator abstraction can also be used for hyper-parameter tuning. The hyper-parameter tuning can be treated as a special case of neural model mutation.

DeepArchitect [46] also strives to decouple model spaces and exploration strategies. Compared to DeepArchitect, Retiarii differentiates itself with the Mutator abstraction. As shown in §7, Retiarii can implement multiple model spaces using a few mutators, demonstrating great reusability and composability. More importantly, with the Mutator abstraction, Retiarii is able to exploit cross-model optimizations easily, which is not addressed previously.

Graph Optimization for Deep Learning. Recently, there are many proposals to optimize the computation of a single neural network model by optimizing the data-flow graph, *e.g.,* TVM [15], DLVM [63], TensorFlow-XLA [38], TASO [31], TensorFlow-Fold [41]. In contrast, Retiarii exploits the cross-model optimizations exposed by Mutator. HiveMind [45], FLEET [26] and some other works [40] apply common sub-expression elimination in the AutoML scenario to deduplicate the common prefix nodes of multiple graphs. This can be considered a special case in Retiarii's larger optimization space, which includes other techniques like operator batching and weight sharing.

9 Conclusion

We propose Retiarii, the first deep learning framework that supports the exploratory training on a neural network model space, rather than on a single neural network model. The core of Retiarii is the Mutator abstraction, which not only allows the specification of different neural network model spaces, interacts with various model exploration strategies, and exposes the model correlations for further optimization, but also serves as a clean interface to separate the three. The design leads to ease of programming, reuse of model space, exploration strategy, and cross-model optimization. Our evaluation demonstrates the effectiveness of the design, showing more than 8× improvement in the overall exploratory-training performance over approaches that rely on existing deep learning frameworks, which only support one model at a time. The artifacts of Retiarii are available at `https://github.com/microsoft/nni/tree/retiarii_artifact`.

Acknowledgments

We thank anonymous reviewers and our shepherd, Prof. Byung-Gon Chun, for their extensive suggestions. We thank Jim Jernigan and Kendall Martin from the Microsoft Grand Central Resources team for providing GPUs for the evaluation of Retiarii. We also thank our colleagues at Microsoft, for their help in implementing and deploying Retiarii: Chengmin Chi (STCA), Shinai Yang (STCA), Deshui Yu (STCA), Chuanjie Liu (STCA). Fan Yang thanks the late Pearl, his beloved cat, for her faithful companion during writing this paper.

References

[1] ast – Abstract Syntax Trees. `https://docs.python.org/3/library/ast.html`, 2020. Online; accessed 30 April 2020.

[2] Kubeflow, The Machine Learning Toolkit for Kubernetes. https://www.kubeflow.org/, 2020. Online; accessed 30 April 2020.

[3] Microsoft open sources breakthrough optimizations for transformer inference on GPU and CPU. https://bit.ly/2xBD70N, 2020. Online; accessed 30 April 2020.

[4] Nas solutions supported by retiarii. https://github.com/microsoft/nni/blob/retiarii_artifact/nas_allstar.md, 2020.

[5] NVIDIA CUDA Multi-Process Service. https://docs.nvidia.com/deploy/pdf/CUDA_Multi_Process_Service_Overview.pdf, 2020. Online; accessed 30 April 2020.

[6] NVIDIA DALI documentation. https://docs.nvidia.com/deeplearning/sdk/dali-developer-guide/index.html, 2020. Online; accessed 30 April 2020.

[7] PyTorch DataLoader. https://pytorch.org/docs/stable/data.html, 2020. Online; accessed 30 April 2020.

[8] The lightweight PyTorch wrapper for ML researchers. https://github.com/PyTorchLightning/pytorch-lightning, 2020. Online; accessed 30 April 2020.

[9] TORCHSCRIPT. https://pytorch.org/docs/stable/jit.html, 2020. Online; accessed 30 April 2020.

[10] Using BERT to extract fixed feature vectors (like ELMo). https://github.com/google-research/bert, 2020. Online; accessed 30 April 2020.

[11] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, 2016.

[12] Gabriel Bender. Understanding and simplifying one-shot architecture search. 2019.

[13] Han Cai, Jiacheng Yang, Weinan Zhang, Song Han, and Yong Yu. Path-level network transformation for efficient architecture search. *arXiv preprint arXiv:1806.02639*, 2018.

[14] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274*, 2015.

[15] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. TVM: An automated end-to-end optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, 2018.

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

[17] Xiangxiang Chu, Bo Zhang, Jixiang Li, Qingyuan Li, and Ruijun Xu. Scarletnas: Bridging the gap between scalability and fairness in neural architecture search. *arXiv preprint arXiv:1908.06022*, 2019.

[18] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *IEEE CVPR 2009*.

[19] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[20] Tobias Domhan, Jost Tobias Springenberg, and Frank Hutter. Speeding up automatic hyperparameter optimization of deep neural networks by extrapolation of learning curves. In *Twenty-Fourth International Joint Conference on Artificial Intelligence*, 2015.

[21] Nick Erickson, Jonas Mueller, Alexander Shirkov, Hang Zhang, Pedro Larroy, Mu Li, and Alexander Smola. Autogluon-tabular: Robust and accurate automl for structured data. *arXiv preprint arXiv:2003.06505*, 2020.

[22] Matthias Feurer, Aaron Klein, Katharina Eggensperger, Jost Tobias Springenberg, Manuel Blum, and Frank Hutter. Auto-sklearn: efficient and robust automated machine learning. In *Automated Machine Learning*, pages 113–134. Springer, 2019.

[23] Adam Gaier and David Ha. Weight agnostic neural networks. In *Advances in Neural Information Processing Systems*, pages 5365–5379, 2019.

[24] Daniel Golovin, Benjamin Solnik, Subhodeep Moitra, Greg Kochanski, John Karro, and D Sculley. Google vizier: A service for black-box optimization. In *Proceedings of the 23rd ACM SIGKDD international conference on knowledge discovery and data mining*, pages 1487–1495, 2017.

[25] Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. Accurate, large minibatch sgd: Training imagenet in 1 hour. *arXiv preprint arXiv:1706.02677*, 2017.

[26] Hui Guan, Laxmikant Kishor Mokadam, Xipeng Shen, Seung-Hwan Lim, and Robert Patton. FLEET: Flexible efficient ensemble training for heterogeneous deep neural networks. *MLSys 2020*, 2020.

[27] Zichao Guo, Xiangyu Zhang, Haoyuan Mu, Wen Heng, Zechun Liu, Yichen Wei, and Jian Sun. Single path one-shot neural architecture search with uniform sampling. *arXiv preprint arXiv:1904.00420*, 2019.

[28] Neil Houlsby, Andrei Giurgiu, Stanislaw Jastrzebski, Bruna Morrone, Quentin De Laroussilhe, Andrea Gesmundo, Mona Attariyan, and Sylvain Gelly. Parameter-efficient transfer learning for nlp. *arXiv preprint arXiv:1902.00751*, 2019.

[29] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*, 2017.

[30] Max Jaderberg, Valentin Dalibard, Simon Osindero, Wojciech M Czarnecki, Jeff Donahue, Ali Razavi, Oriol Vinyals, Tim Green, Iain Dunning, Karen Simonyan, et al. Population based training of neural networks. *arXiv preprint arXiv:1711.09846*, 2017.

[31] Zhihao Jia, Oded Padon, James Thomas, Todd Warszawski, Matei Zaharia, and Alex Aiken. TASO: optimizing deep learning computation with automatic generation of graph substitutions. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 47–62, 2019.

[32] Haifeng Jin, Qingquan Song, and Xia Hu. Auto-keras: An efficient neural architecture search system. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 1946–1956, 2019.

[33] Kirthevasan Kandasamy, Willie Neiswanger, Jeff Schneider, Barnabas Poczos, and Eric P Xing. Neural architecture search with bayesian optimisation and optimal transport. In *Advances in Neural Information Processing Systems*, pages 2016–2025, 2018.

[34] Viggo Kann. On the approximability of the maximum common subgraph problem. In *Annual Symposium on Theoretical Aspects of Computer Science*, pages 375–388. Springer, 1992.

[35] Nitish Shirish Keskar, Dheevatsa Mudigere, Jorge Nocedal, Mikhail Smelyanskiy, and Ping Tak Peter Tang. On large-batch training for deep learning: Generalization gap and sharp minima. *arXiv preprint arXiv:1609.04836*, 2016.

[36] Scott Kirkpatrick, C Daniel Gelatt, and Mario P Vecchi. Optimization by simulated annealing. *science*, 220(4598):671–680, 1983.

[37] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[38] Chris Leary and Todd Wang. XLA: Tensorflow, compiled. *TensorFlow Dev Summit*, 2017.

[39] Hanxiao Liu, Karen Simonyan, and Yiming Yang. Darts: Differentiable architecture search. *arXiv preprint arXiv:1806.09055*, 2018.

[40] Rui Liu, Sanjan Krishnan, Aaron J Elmore, and Michael J Franklin. Understanding and optimizing packed neural network training for hyper-parameter tuning. *arXiv preprint arXiv:2002.02885*, 2020.

[41] Moshe Looks, Marcello Herreshoff, DeLesley Hutchins, and Peter Norvig. Deep learning with dynamic computation graphs. *arXiv preprint arXiv:1702.02181*, 2017.

[42] Ningning Ma, Xiangyu Zhang, Hai-Tao Zheng, and Jian Sun. Shufflenet v2: Practical guidelines for efficient cnn architecture design. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 116–131, 2018.

[43] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.

[44] Steven Muchnick et al. *Advanced compiler design implementation*. Morgan kaufmann, 1997.

[45] Deepak Narayanan, Keshav Santhanam, Amar Phanishayee, and Matei Zaharia. Accelerating deep learning workloads through efficient multi-model execution. In *NIPS Workshop on Systems for Machine Learning (December 2018)*, 2018.

[46] Renato Negrinho, Matthew Gormley, Geoffrey J Gordon, Darshan Patil, Nghia Le, and Daniel Ferreira. Towards modular and programmable architecture search. In *Advances in Neural Information Processing Systems*, pages 13715–13725, 2019.

[47] Randal S Olson and Jason H Moore. Tpot: A tree-based pipeline optimization tool for automating machine learning. In *Automated Machine Learning*, pages 151–160. Springer, 2019.

[48] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al.

PyTorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems*, pages 8024–8035, 2019.

[49] Matthew E Peters, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, and Luke Zettlemoyer. Deep contextualized word representations. *arXiv preprint arXiv:1802.05365*, 2018.

[50] Hieu Pham, Melody Y. Guan, Barret Zoph, Quoc V. Le, and Jeff Dean. Efficient neural architecture search via parameter sharing. *CoRR*, abs/1802.03268, 2018.

[51] Esteban Real, Alok Aggarwal, Yanping Huang, and Quoc V Le. Regularized evolution for image classifier architecture search. In *Proceedings of the aaai conference on artificial intelligence*, volume 33, pages 4780–4789, 2019.

[52] Sylvestre-Alvise Rebuffi, Hakan Bilen, and Andrea Vedaldi. Efficient parametrization of multi-domain deep neural networks. In *IEEE CVPR 2018*.

[53] Sylvestre-Alvise Rebuffi, Hakan Bilen, and Andrea Vedaldi. Learning multiple visual domains with residual adapters. In *Advances in Neural Information Processing Systems*, pages 506–516, 2017.

[54] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *IEEE CVPR 2018*.

[55] Richard Socher, Alex Perelygin, Jean Wu, Jason Chuang, Christopher D Manning, Andrew Y Ng, and Christopher Potts. Recursive deep models for semantic compositionality over a sentiment treebank. In *Proceedings of the 2013 conference on empirical methods in natural language processing*, pages 1631–1642, 2013.

[56] Francisco J Solis and Roger J-B Wets. Minimization by random search techniques. *Mathematics of operations research*, 6(1):19–30, 1981.

[57] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. In *IEEE CVPR 2016*.

[58] Kai Sheng Tai, Richard Socher, and Christopher D Manning. Improved semantic representations from tree-structured long short-term memory networks. *arXiv preprint arXiv:1503.00075*, 2015.

[59] Mingxing Tan, Bo Chen, Ruoming Pang, Vijay Vasudevan, and Quoc V. Le. Mnasnet: Platform-aware neural architecture search for mobile. *CoRR*, abs/1807.11626, 2018.

[60] Mingxing Tan and Quoc V Le. Efficientnet: Rethinking model scaling for convolutional neural networks. *arXiv preprint arXiv:1905.11946*, 2019.

[61] Chris Thornton, Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. Auto-weka: Combined selection and hyperparameter optimization of classification algorithms. In *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 847–855, 2013.

[62] Yujing Wang, Yaming Yang, Yiren Chen, Jing Bai, Ce Zhang, Guinan Su, Xiaoyu Kou, Yunhai Tong, Mao Yang, and Lidong Zhou. Textnas: A neural architecture search space tailored for text representation. *arXiv preprint arXiv:1912.10729*, 2019.

[63] Richard Wei, Lane Schwartz, and Vikram Adve. DLVM: A modern compiler infrastructure for deep learning systems. *arXiv preprint arXiv:1711.03016*, 2017.

[64] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu Zhang, et al. Gandiva: Introspective cluster scheduling for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 595–610, 2018.

[65] Chris Ying, Aaron Klein, Esteban Real, Eric Christiansen, Kevin Murphy, and Frank Hutter. Nas-bench-101: Towards reproducible neural architecture search. *arXiv preprint arXiv:1902.09635*, 2019.

[66] Hang Zhang, Kristin Dana, Jianping Shi, Zhongyue Zhang, Xiaogang Wang, Ambrish Tyagi, and Amit Agrawal. Context encoding for semantic segmentation. In *IEEE CVPR 2018*.

[67] Hongpeng Zhou, Minghao Yang, Jun Wang, and Wei Pan. Bayesnas: A bayesian approach for neural architecture search. *arXiv preprint arXiv:1905.04919*, 2019.

[68] Jinan Zhou, Andrey Velichkevich, Kirill Prosvirov, Anubhav Garg, Yuji Oshima, and Debo Dutta. Katib: A distributed general automl platform on kubernetes. In *2019 USENIX Conference on Operational Machine Learning (OpML 19)*, pages 55–57, 2019.

[69] Barret Zoph and Quoc V Le. Neural architecture search with reinforcement learning. *arXiv preprint arXiv:1611.01578*, 2016.

[70] Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V. Le. Learning transferable architectures for scalable image recognition. *CoRR*, abs/1707.07012, 2017.

A Artifact Appendix

A.1 Abstract

This artifact is designed to reproduce the main results of this work, which have two goals:

- **Functionality**: Retiarii can express NAS spaces using mutators, explore spaces using Exploration Engine, and accelerate the exploration using cross-model optimization.
- **Performance**: Retiarii's cross-model optimization achieves the performance number claimed in §7.

A.2 Artifact check-list

- **Algorithm:** yes
- **Data set:** ImageNet [18], SST [55]
- **Run-time environment:** Ubuntu 16.04, CUDA 10.0, cuDNN 7.6.5. Root access is required.
- **Hardware:** GPUs with NVIDIA MPS.
- **Metrics:** training throughput; job completion time; model validation accuracy.
- **Output:** Web UI; stdout from console.
- **Required disk space:** 200 GB
- **Expected experiment run time:** 20 hours
- **Public link:** `https://github.com/microsoft/nni/tree/retiarii_artifact`
- **Code licenses:** MIT License

A.3 Description

A.3.1 How to access

Clone the "retiarii_artifact" branch of Microsoft NNI's GitHub repository.

```
git clone -b retiarii_artifact https://github.com/Microsoft/nni.git
```

A.3.2 Hardware dependencies

This artifact requires at least one server with four NVIDIA V100 GPUs.

A.3.3 Software dependencies

- CUDA 10.1;
- cuDNN 7.6.5;
- Python 3.7;
- NVIDIA DALI;
- NVIDIA Apex;
- PyTorch 1.5.1;
- TensorFlow 2.3;
- Other Python packages in "requirements.txt".

A.3.4 Data sets

- ImageNet: should be placed at "retiarii_perf/data/imagenet".
- SST: The three text files (dev.txt, test.txt, train.txt) SST should be placed at "retiarii_perf/data/sst/trees".

A.4 Installation

For running Retiarii's artifact, please install NNI v1.8 first. This artifact contains two parts. In the folder of "retiarii_nas", we demonstrate the functionality of Retiarii to express different NAS solutions. In the folder of "retiarii_perf", we evaluate Retiarii's performance using cross-model optimization.

For some experiments, it requires NVIDIA MPS to be enabled. To start NVIDIA MPS:

```
sudo ./mps_scripts/init_mps_for_all_gpus.sh
./mps_scripts/set_mps_env_for_all_gpus.sh
```

To stop NVIDIA MPS:

```
sudo ./mps_scripts/stop_mps_for_all_gpus.sh
```

A.5 Evaluation: NAS Solution All-stars

In the folder of "retiarii_nas", we have implemented 17 NAS solutions using Retiarii. We support both PyTorch and TensorFlow. Weight Agnostic Networks (wann), Path-level NAS (path_level), and Hierarchical Representation (hierarchical) are implemented with TensorFlow. Other NAS solutions are implemented with PyTorch. We also provide a script to test them, which can be started using the following command.

```
python3 retiarii.py e2e_launch.py [nas_mode]
```

(Use "python3 retiarii.py -L" to get the list of supported models)

After the command is executed, a Web UI URL will be given, which contains the trial execution status.

Note that, to speed up the test, we run each generated model by only one mini-batch (thus, returned values are all 0), you are free to remove the 'break's in e2e_launch.py (ModelTrain, ModelTrainCifar, ModelTrainTextNAS) to run each generated model completely.

This artifact has supported three classic exploration strategies: random, reinforcement learning, and evolution, and also has supported two differentiable training strategies: DARTS training strategy and ProxylessNAS training strategy. Other exploration strategies have been supported in NNI (`https://github.com/microsoft/nni/blob/retiarii_artifact/backend_nni/docs/en_US/Tuner/BuiltinTuner.md`), have not been integrated into this artifact. They will be formally supported in Retiarii official release.

Paper Claim: Retiarii is able to support 27 NAS solutions.

Clarification: We have included 17 of the 27 NAS solutions in the artifact evaluation. The remaining ones only have minor differences with the included implementations (e.g., EfficientNet v.s. MnasNet, SCARLET-NAS v.s. FairNAS v.s. SPOS). We believe the included ones are sufficient to demonstrate the programmability of Retiarii. Full support of the 27 NAS solutions will be included in an official release version of Microsoft NNI.

A.6 Evaluation: Retiarii Performance

A.6.1 Micro-benchmark: Deduplication to avoid CPU bottleneck

Execution command:

```
python artifact_start.py micro_dedup_cpu --n=8
```

This python script will start 8 jobs (each GPU runs two jobs), then profile the total throughput. This command takes 1.5 minutes. The result will be printed after the profiling as follows. The error should be within 10%.

```
Throughput: 4746.849792184445 samples/s
```

Paper Claim: In Figure 12, when running 8 models, Retiarii can achieve about 5000 samples/s.

A.6.2 Micro-benchmark: Deduplication to avoid GPU bottleneck

Execution command:

```
python artifact_start.py micro_dedup_gpu --n=12
```

This python script will start 12 jobs. GPU-0 runs one job, each of the other three GPUs run 4 jobs). Then it profiles the total throughput. This command takes 1.5 minutes. The result will be printed after the profiling as follows. The error should be within 10%.

```
Throughput: 5028.187640607402 samples/s
```

Paper Claim: In Figure 13, when running 12 models, Retiarii can achieve about 5100 samples/s.

A.6.3 Micro-benchmark: Operator batching

Execution command:

```
python artifact_start.py micro_batching --n=192
```

This python script will use Retiarii to pack 192 models into one job to be run on a single GPU-0. Then it profiles the total throughput. This command takes 10 minutes. It is normal if it has no output for a long time, because it takes 3 minutes for the cross-model optimization policy to calculate a plan. The result will be printed after the profiling as follows. The error should be within 10%.

```
Throughput: 6124.981150684514 samples/s
```

Paper Claim: In Figure 14, when batching 192 models, Retiarii can achieve about 5800 samples/s.

A.6.4 End-to-end Evaluation: MnasNet using DALI

Execution command:

```
python artifact_start.py e2e_dali_mnasnet
```

This experiment will train 1000 MnasNet models in 20 batches (each batch has 50 models). Each model will be trained for 1 epoch on ImageNet, which will be very time-consuming and costly if we train all 1000 models. Since we only want to know the training time but not the validation accuracy. We use a workaround to "fast-forward" the training. We profile each job for 150 mini-batches to measure the iteration time. Then we use the measured job speed to emulate the experiment with a simple job scheduler (implemented in "fast_scheduler.py"). The experiment takes about 1 hour to run. The result will be printed as follows. The error should be within 10%.

```
124.35633072276445 hours for mnasnet w/ BS=32
```

Paper Claim: In Figure 15(a), when Batch Size=32, Retiarii can finish NAS exploration of MnasNet in about 130 hours.

A.6.5 End-to-end Evaluation: SPOS training using mixed parallelism

Execution command:

```
python artifact_start.py e2e_spos_mix_parallel
  --n=4
```

This python script will start 4 jobs, each on one GPU, to train SPOS in mixed parallelism, a new type of training parallelism we proposed for weight sharing-based training. The super-graph is generated in the function "_gen_spos_super_graph(n_job)" in "artifact_start.py". In the paper, we used 8 V100 GPUs in two servers, which takes about 7.45 hours to train SPOS for 60 epochs achieving 61.2% average validation accuracy. The result will be printed as follows.

```
[03/31 02:40:46] INFO (main) Epoch [60/60]
    Validation Step [196/196]  acc1 0.650000
    (0.611117)  acc5 0.887500 (0.833490)  loss
    2.359303 (2.586974)
```

Note that, the training of SPOS is unstable. The average validation accuracy could vary from 60% to 62%. For your reference, we also provide the training log we obtained on eight V100 GPUs in "data/spos_8_v100.log".

Paper Claim: In Figure Figure 17, Retiarii's mixed parallelism can train SPOS for 60 epochs with a batch size of 256 to achieve 61.11%.

A.7 Experiment customization

New experiments can be customized and added in "retiarii_nas/e2e_launch.py" and "retiarii_perf/artifact_start.py".

A.8 Notes

NVIDIA CUDA MPS may fail if a job is not stopped properly, which requires NVIDIA CUDA MPS to be restarted. Experiments in "retiarii_nas" will kill a running job for saving time, but may trigger the failure of NVIDIA CUDA MPS. We suggest to disable NVIDIA CUDA MPS when running experiments in "retiarii_nas".