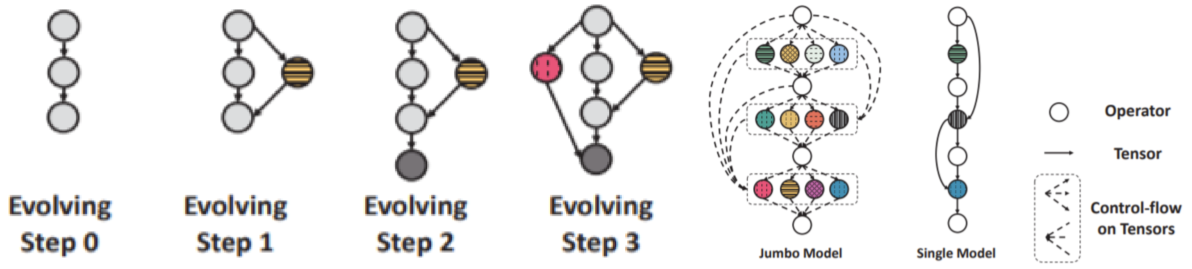
Summary of “Retiarii: A Deep Learning Exploratory-Training Framework”
Tianyi Ge (gty), Lingyun Guo (linguo), Haojie Ye (yehaojie)

Problem and Motivation

Retrofitting exploratory-training in the training process is cumbersome in the current ML frameworks PyTorch and Tensorflow. Devising a model for a particular task (exploring a “jumbo” model, tweaking a model, etc. shown in the below figure) 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.



There are no state-of-the-art approaches for exploratory-training currently. A programmer 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. The implementation often tightly couples an exploration strategy with a specific model space, making it very difficult to be applicable to different model spaces. For example, the implementations embedded in the jumbo models are hardly reusable by other model spaces. Retiarii advocates a clean separation of concerns and strives for simplicity and modularity in generating tweaked DNN models for exploratory-training automatically.

Hypothesis

We can introduce the *Mutator* abstraction as the basis for specifying a DNN model space and for defining an exploration strategy. A DNN model space for an exploratory-training process can be defined as a set of base models and a set of mutators. Then the DNN mode space becomes the base models plus any subsequent models produced by applying mutators.

Solution Overview

Retiarii represents a DNN model as a data-flow graph (DFG), in which each node represents an operator or a subgraph with one or multiple input and output tensors and each edge connects an output of a node to an input of another node. It 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. It can simply import base models defined in an existing deep learning framework such as TensorFlow. Retiarii uses a mutator to connect the specifications of DNN model spaces and exploration strategies. Each mutator identifies subgraphs of a DFG of a target model to operate on with a matching criteria and modifies the matched subgraphs to create a new model by a series of graph construction operations. Retiarii records all the mutation primitives called in a mutator. Therefore, it can easily identify model correlations across

instantiated 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.
In addition, Retiarii's just-in-time (JIT) engine, takes as input one or more base models, a set of mutators, and a policy describing the exploration strategy, instantiate models to explore on the fly and manage the training of instantiated models dynamically.
Retiarii specifies three cross-model optimizations:

- Common Sub-expression Elimination (CSE). This approach is useful for merging prefix nodes of a DFG, because they are often non-trainable operators for data loading and preprocessing.
- Operator Batching. Common operators with different inputs and weights can potentially be batched together and computed in a single operator kernel.
- Super-Graph for Weight Sharing. Using shared weights inherited from other graphs to continue training, 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.

Retiarii introspectively selects graphs to merge. Moreover, it specifically optimizes device placement of CSE-optimized graphs and training parallelism. In Retiarii, all cross-graph optimizations are applied within every batch of models. It greedily packs as many models as possible into one GPU.
Retiarii also takes the advantage of model parallel training of the super-graph of all models: when the super-graph of all models is too large to fit into one GPU, it splits the super-graph onto multiple GPUs. Retiarii spreads the instantiated models into multiple super-graphs (each on a GPU) to be trained together.

Limitations and Possible Improvements
Retiarii has a main limitation that it does not support dynamic graphs. Retiarii's mutators are applied to a static base model. The author thinks it is too difficult to extract a graph representation from a highly dynamic base model like LSTMs. Also, the current implementation of operator batching is limited, and more advanced batching requires implementing new GPU kernels. When a model is mutated, it requires additional programming efforts for the programmer to match the shape of adjacent layers' input/output tensors. This adds an additional burden to the programmer for fully automatically adopting the generated new model with tweaks.

Summary of Class Discussion
Q: The paper provided a lot of details of their approach and fully displayed why their approach is better, but the central idea may not be explicit.
A: The paper introduced Retiarii as the first framework that supports deep learning exploratory-training. It does not have a detailed example of policy of making mutations, but mainly explains the contributions of Retiarii and verified its overall efficiency.

Summary of “Fluid: Resource-aware Hyperparameter Tuning Engine”

Tianyi Ge (gty), Lingyun Guo (linguo), Haojie Ye (yehaojie)

Problem and Motivation

Hyperparameter tuning is an optimization loop in order to find the best set of hyperparameters that are likely to produce the highest validation accuracy. But the state-of-the-art approach of parameter tuning often is executed on the GPU without considering the possibility of intra-worker and inter-worker sharing, failing to make good use of the available resources. For example, there can be more workers than training trials, which can lead to resource underutilization if each training trial only uses one worker. Moreover, a single training trial may not fully occupy one worker; the single-trial-to-single-worker mapping then leaves room to further improve resource utilization when there are more training trials than workers.
Existing hyperparameter tuning algorithms suffer from two major problems:

1. GPU compute resources are under-utilized. During hyperparameter tuning, not all GPUs are activated. Also when a GPU is used, the overall utilization is at most 60%, suggesting that a single trial often cannot fully saturate the GPU. Multiple trials can be stacked/packed together to reduce the average trial completion time.
2. Asynchronous tuning algorithm fully utilizes the GPU but does not produce useful work. It is observed that as we increase the training trial concurrency, the best configuration is not necessarily identified faster. More GPU seconds are consumed to reach the same search target when the asynchronous tuning scales out.

Hypothesis

Hyperparameter tuning algorithms can be converted into a separate execution engine. By abstracting hyperparameter tuning jobs as a sequence of TrialGroups, we can provide a generic interface for hyperparameter tuning. The hyperparameter tuning process then executes in a decoupled fashion with the training logic, and gets scheduled by considering both the current workload and available resources. This could improve utilization and speed up the tuning process.

Solution Overview

The paper proposed Fluid, a generalized execution engine for hyperparameter tuning which can allocate resources to each training trial for efficient resource usage and minimized makespan. Fluid designs the execution plan for each trial based on the current training workload, the evaluation plan and the current resource status. Also, Fluid reports the trial results back to the tuning algorithm for the next step of decision. Fluid aims at solving three challenges: to adapt to a wide variety of tuning algorithms, to deal with highly dynamic training workloads and resources, and to consider heterogeneity in training trial profiles.

TrialGroup abstraction is a group of trials with respective training budget, which is designed to generalize different hyperparameter tuning algorithms i.e. all kinds of hyperparameter tuning algorithms could be expressed by a sequence of TrialGroup so that Fluid can execute them in a uniform manner. Trials can be either grid/random search (unordered) or sequential algorithms.

Unlike grid/random search, sequential model-based algorithms require the trials to be wrapped into multiple ordered TrialGroups.

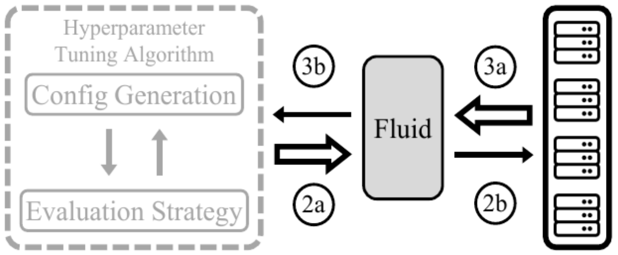
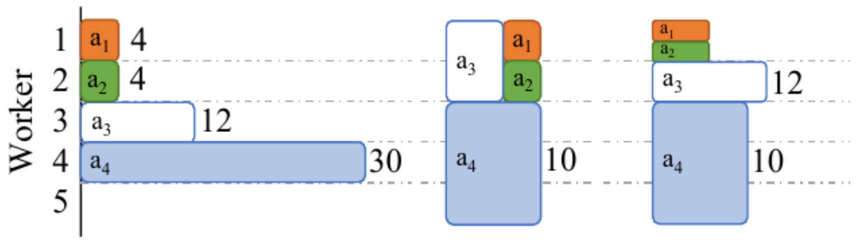


Figure 4. With Fluid, the tuning algorithm (2a) submits **TrialGroup** according to its evaluation strategy and (3b) gets feedbacks back anytime they are available. Fluid itself manages (2b) the job execution and handles (3a) resource changing events.

The action of Fluid will only be triggered when new TrialGroups are added or some resources are freed. Fluid introduces two types of algorithms for resource allocation

1. StaticFluid: schedule new TrialGroups onto idle resources
2. DynamicFluid: reallocate resources to mitigate multiple sources of overheads

The entire problem can be abstracted to a strip packing problem by considering each trial within a group as a rectangle (strip). The two major metrics: 1) worker (GPU) number and 2) the makespan are the two dimensions.



(a) Default (b) Optimal (c) Fluid

This problem reduction relies on the following two major types of resource parallelism:

1. intra-GPU parallelism (Nvidia MPS): execute more trials with under-utilized resources
2. inter-GPU parallelism (Distributed training): fully utilize the idle resources

StaticFluid allocates resources based on the ratio of each trials's runtime (height) to the sum of runtime under the limit of maximum intra-GPU and intel-GPU sharing.

Algorithm 1 StaticFluid

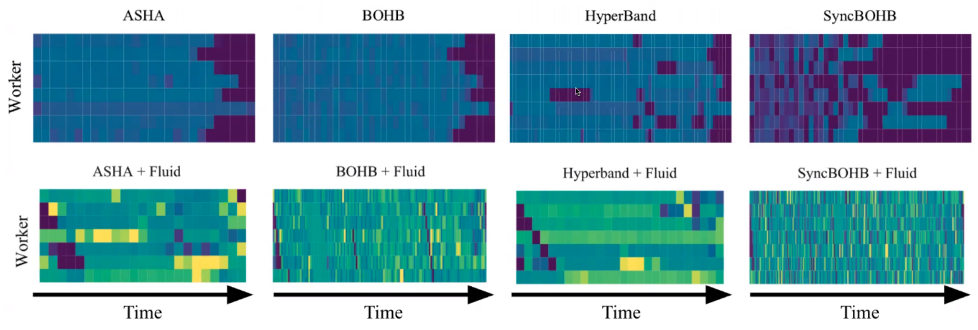
```
1: def StaticFluid(TrialGroup $A$, Idle Resources $M'$)
2:     Sort $a_i$ by $h_{i,1}$ in non-increasing order
3:     for all $a_i \in A$ do
4:         $w_i = \min(\max(\lfloor \frac{h_{i,1}}{\sum_j h_{j,1}} n \rfloor, \frac{1}{c}), d)$
5:         Allocate $a_i$ with $w_i$ resources
```

Algorithm 2 DynamicFluid

```
1: def DynamicFluid(TrialGroup $A$, Total Res. $M$)
2:     Sort $a_i$ by $h_{i,1}$ in non-increasing order
3:     for all $a_i \in A$ do
4:         $w_i' = \min(\max(\lfloor \frac{h_{i,1}}{\sum_j h_{j,1}} n \rfloor, \frac{1}{c}), d)$
5:         if $w_i' > w_i$ and $h_{i,w_i'} + \epsilon < h_{i,w_i}$
6:             Update $a_i$ with $w_i$ resources        ▷ Scale up
7:         else if $w_i' < w_i$ and $w_i'(h_{i,w_i'} + \epsilon) < w_i h_{i,w_i}$
8:             Update $a_i$ with $w_i$ resources        ▷ Scale down
```

DynamicFluid, in addition to StaticFluid, checks the overhead of scale up and scale down so that it updates the allocation without leading to performance degradation.

For evaluations, the paper benchmarked ASHA, BOHB, Hyperband and SyncBOHB w/ and w/o Fluid. The results show that not only Fluid improves the makespan and resource utilization, it also raises the best validation accuracy. The makespan is reduced by from 10% to 100%, and the utilization is also greatly optimized, regardless of the worker number.



Limitations and Possible Improvements

Fluid is currently an intermediate component between GPU and hyperparameter tuning algorithms, but it can be more convenient to use if provided as a class to use within tuning algorithms. Also there might be some trials that can indicate the necessity of future trials. But if it's scheduled later it might not be that efficient, which can potentially be used to save unnecessary executions. Fluid is not able to detect that trial feature so far.

Summary of Class Discussion

Q: How do you measure average utilization within a single gpu? Is it how much the computer is being used or memory is being used or the memory bandwidth or a combination of them?
A: Computation utilization.

Q: It's very hard to tell from the error bar so do you know, like is it always better are there are cases where actually adding fluid is worse?
A: Hard to conclude for other untested cases.

Comments:

1. Fluid can be rewritten to a class for ease of use.
2. There might be some trials that can indicate the necessity of future trials. But if it's scheduled later it might not be that efficient.
3. Keep track of the training time along with the configuration so that the execution of future trials with similar configurations can be estimated