Tune: A Research Platform for Distributed Model Selection and Training

Richard Liaw*   RLIAW@BERKELEY.EDU

Eric Liang*   ERICLIANG@BERKELEY.EDU

Robert Nishihara   RKN@BERKELEY.EDU

Philipp Moritz   PCM@BERKELEY.EDU

Joseph E. Gonzalez   JEGONZAL@EECS.BERKELEY.EDU

Ion Stoica   ISTOICA@CS.BERKELEY.EDU

Abstract

Modern machine learning algorithms are increasingly computationally demanding, requiring specialized hardware and distributed computation to achieve high performance in a reasonable time frame. Many hyperparameter search algorithms have been proposed for improving the efficiency of model selection, however their adaptation to the distributed compute environment is often ad-hoc. We propose Tune, a unified framework for model selection and training that provides a narrow-waist interface between training scripts and search algorithms. We show that this interface meets the requirements for a broad range of hyperparameter search algorithms, allows straightforward scaling of search to large clusters, and simplifies algorithm implementation. We demonstrate the implementation of several state-of-the-art hyperparameter search algorithms in Tune. Tune is available at http://ray.readthedocs.io/en/latest/tune.html.

1. Introduction

Machine learning pipelines are growing in complexity and cost. In particular, the model selection stage, which includes model training and hyperparameter tuning, can take the majority of a machine learning practitioner's time and consume vast amounts of computational resources. Take for example a researcher aiming to train ResNet-101, a convolutional neural-network model with millions of parameters. Training this model can take around 24 hours on a single GPU, and performing model selection sequentially will take weeks to complete. Naturally, one would be inclined to train the model on a cluster in a distributed fashion (Goyal et al. (2017)) and utilize many machines to perform model selection in parallel.

To this end, the research community has developed numerous techniques for accelerating model selection including those that are sequential (Snoek et al. (2012)), parallel (Li et al. (2016)), and both (Jaderberg et al. (2017)). However, each technique is often implemented on its own, tied to a particular framework, is closed source, or perhaps not even reproducible without significant computational resources (Zoph and Le (2016)). Further, often times these techniques require significant investment in software infrastructure for the execution of experiments.
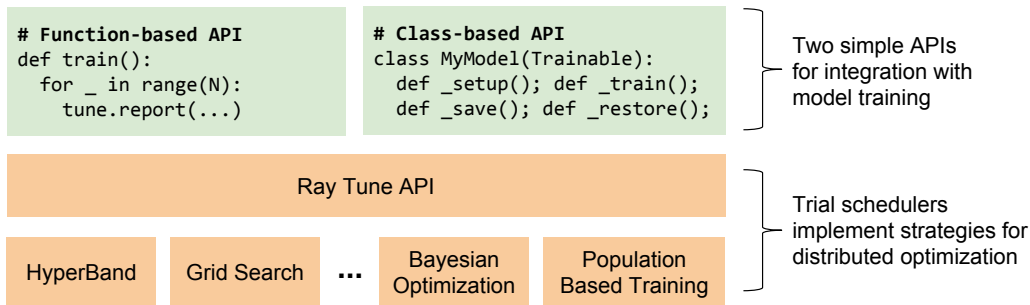
```
# Function-based API        # Class-based API
def train():                class MyModel(Trainable):
  for _ in range(N):          def _setup(); def _train();
    tune.report(...)          def _save(); def _restore();
```

Two simple APIs
for integration with
model training

Ray Tune API

HyperBand   Grid Search   ...   Bayesian   Population
                                 Optimization   Based Training

Trial schedulers
implement strategies for
distributed optimization

Figure 1: Tune provides narrow-waist interfaces that training scripts can implement with a few lines of code changes. Once done, this enables the use of Tune for experiment management, result visualization, and a choice of trial scheduling strategies. This narrow interface also enables *AutoML researchers* to easily swap out different search algorithms for comparison, or release distributed implementations of new algorithms without needing to worry about distributed scaffolding.

The contributions of this paper are as follows:

1. We introduce Tune, an open source framework for distributed model selection.
2. We show how Tune's APIs enable the easy reproduction and integration of a wide variety of state-of-the-art hyperparameter search algorithms.

2. Related work

There are multiple open source systems for model selection.

HyperOpt, Spearmint, and HPOLib (Snoek et al. (2012); Eggensperger et al. (2013)) are distributed model selection tools that manage both the search and evaluation of the model, implementing search techniques such as random search and tree of parzen estimators (TPE). However, both frameworks are tightly coupled to the search algorithm structure and requires manual management of computational resources across a cluster. Further, systems such as Spearmint, HyperOpt, and TuPAQ (MLBase) (Sparks et al. (2015)) treat a full trial execution as an atomic unit, which does not allow for intermediate control of trial execution. This inhibits efficient usage of cluster resources and also does not provide the expressivity needed to support algorithms such as HyperBand.

Google Vizier (Golovin et al. (2017)) is a Google-internal service that provides model selection capabilities. Similar to Tune, Vizier provides parallel evaluation of trials, hosts many state-of-the-art optimization algorithms, provides functionality for performance analysis. However, it is first and foremost a service and is tied to closed-source infrastructure.

Mistique (Vartak et al. (2018)) is also a system that addresses model selection. However, rather than focusing on the execution of the selection process, Mistique focuses on model debugging, emphasizing iterative procedures and memory footprint minimization.

Finally, Auto-SKLearn (Feurer et al. (2015)) and Auto-WEKA (Thornton et al. (2013)) are systems for automating model selection, integrating meta learning and ensembling into a single system. The focus of these systems is at the execution layer rather than the algorithmic level. This implies that in principle, it would be possible to implement Auto-WEKA and Auto-SKLearn on top of Tune, providing distributed execution of these AutoML components. Further, both Auto-SKLearn and Auto-WEKA are tied to Scikit-Learn and WEKA respectively as the only machine learning frameworks supported.

3. Requirements for API generality

We refer to a *trial* as a single training run with a fixed initial hyperparameter configuration. An *experiment* (similar to a "Study" in Vizier), is a collection of trials supervised by Tune using one of its trial scheduling algorithms (which implement model selection).

A platform for model search and training blends both sequential and parallel computation. During search, many trials are evaluated in parallel. Hyperparameter search algorithms examine trial results in sequence and make decisions that affect the parallel computation. In order to support both a broad range of training workloads and selection algorithms, a framework for model search needs to meet the following requirements:

- Ability to handle irregular computations: Trials often vary in length and resource usage.
- Ability to handle resource requirements of arbitrary user code and third-party libraries. This includes parallelism and the use of hardware resources such as GPUs.
- Ability to make search / scheduling decisions based on intermediate trial results. For example, genetic algorithms commonly clone or mutate model parameters in the middle of training. Algorithms that perform early stopping also use intermediate results to make stopping decisions.

For a good user experience, the following features are also necessary:

- The monitoring and visualization of trial progress and outcomes.
- Simple integration and specification of the experiment to execute.

To meet these requirements, we propose the Tune user-facing and scheduling APIs (Section 4) and implement it on the Ray distributed computing framework (Moritz et al. (2017)). The Ray framework provides the underlying distributed execution and resource management. Its flexible task and actor abstractions allow Tune to schedule irregular computations and make decisions based on intermediate results.

4. Tune API

Tune provides two development interfaces: a *user API* for users seeking to train models and a *scheduling API* for researchers interested in improving the model search process itself. As a consequence of this division, users of Tune have a choice of many search algorithms. Symmetrically, the scheduler API enables researchers to easily target a diverse range of workloads and provides a mechanism for making their algorithms available to users.

4.1. User API

Model training scripts are commonly implemented as a loop over a model improvement step, with results periodically logged to the console (e.g., every training epoch). To support the full range of model search algorithms, Tune requires access to intermediate training results, the ability to snapshot training state, and also the ability to alter hyperparameters in the middle of training. These requirements can be met with minimal modifications to existing user code via a *cooperative* control model.

```python
def my_train_func(tune):
  model = NeuralNet(tune.params["activation"])
  optimizer = torch.optim.SGD(
    model.parameters(),
    lr=tune.params["lr"],
    momentum=tune.params["momentum"])
  dataset = ( ... )
  for idx, (data, target) in enumerate(dataset):
    if tune.should_checkpoint():
      tune.record_checkpoint(file=model.checkpoint())
      # ...
    optimizer.step()
    tune.report(
      iteration=idx, accuracy=eval_accuracy(...))
```

(a) Function-based API

```python
class MyTrainableModel:
  def __init__(self, tune_params):
    self.model = NeuralNet(tune_params["activation"])
    self.optimizer = torch.optim.SGD(
      self.model.parameters(), lr=tune_params["lr"],
      momentum=tune_params["momentum"])
    # ...

  def train(self):
    # ...
    self.optimizer.step()
    return tune.Result(
      iteration=self.idx, accuracy=eval_accuracy(...))

  def save(self): ...
  def restore(self, checkpoint_path): ...
```

(b) Class-based API

Figure 2: Tune offers both a function-based *cooperative* control API and a class-based API that allows for direct control over trial execution. Either can be adopted by the user to enable control over model training via Tune's trial schedulers.

Consider a typical model training script as shown in 2(a). A handle to Tune is passed into the function. To integrate with Tune, the model and optimizer hyperparameters are pulled from the **tune.params** map, checkpoints are created when **tune.should_checkpoint()** returns positive, and the saved checkpoint file is passed to **tune.record_checkpoint()**. Intermediate results are reported via **tune.report()**. These cooperative calls enable Tune scheduling algorithms to monitor the training progress of each trial (via the reported metrics), save and clone promising parameters (via checkpoint and restore), and alter hyperparameters at runtime (by restoring a checkpoint with a changed hyperparameter map). Critically, these calls require minimal changes to existing user code.

Tune can also *directly control* trial execution if the user extends the trainable model class (Figure 2(b)). Here, training steps, checkpointing, and restore are implemented as class methods which Tune schedulers call to incrementally train models. This mode of execution has some debuggability advantages over cooperative control; we offer both to users. Internally, Tune inserts adapters over the cooperative interface to provide a facade of direct control to trial schedulers.

4.2. Scheduler API

Given the ability to create trials and control their execution, the next question is how trials should be scheduled. Tune's *trial schedulers* operate over a set of possible trials to run, prioritizing trial execution given available cluster resources. In addition, they can add to the list of trials to execute (e.g., based on suggestions from HyperOpt).

The simplest trial scheduler executes trials sequentially, running each until a stopping condition is met. Trials are launched in parallel when sufficient resources are available in the cluster. However, this is not all trial schedulers can do. They can:

1. Early stop a trial that is not performing well based on its intermediate results.
2. Adjust the annealing of hyperparameters such as learning rate.

3. Clone the parameters of a promising trial and launch additional trials that explore the nearby hyperparameter space.
4. Query a shared database of trial results to choose promising hyperparameters.
5. Prioritize resource allocation between a large number of trials, more than can run concurrently given available resources.

The primary interface for a trial scheduler is as follows:

```python
class TrialScheduler:
    def on_result(self, trial, result): ...
    def choose_trial_to_run(self): ...
```

The interface is event based. When Tune has resources available, it calls **scheduler.choose_trial_to_run()** to get a trial to launch on the cluster. As results for the trial become available, the **scheduler.on_result()** callback is invoked and the scheduler returns an flag indicating whether to continue, checkpoint, stop, or restart a trial with an updated hyperparameter configuration. This interface is sufficient to provide a broad range of hyperparameter tuning algorithms including Median Stopping Rule (Golovin et al. (2017)), Bayesian Optimization approaches (Bergstra et al. (2013)), HyperBand (Li et al. (2016)), and Population-based Training (Jaderberg et al. (2017)).

We note that Tune keeps the metadata for active trials in memory and relies on checkpoints for fault tolerance. This drastically simplifies the design of trial schedulers and is not a limitation in practice. Trial scheduler implementations are free to leverage external storage if necessary.

4.3. Putting it together

To launch an experiment, the user must specify their model training function or class (Figures 1 and 2(b)), an initial set of trials, and a trial scheduler. The following is a minimal example:

```python
def my_func(): ...
tune.run_experiments(my_func, {
    "lr": tune.grid_search([0.01, 0.001, 0.0001]),
    "activation": tune.grid_search(["relu", "tanh"]),
}, scheduler=HyperBand)
```

Here, we use Tune's built-in DSL to specify a small 3×2 grid search over two hyperparameters. These serve as the initial set of trials input to the scheduler. Tune's parameter DSL offers features similar to those provided by HyperOpt (Bergstra et al. (2013)). Alternatively, users can generate the list of initial trial configurations with the mechanism of their choice. Once an experiment is launched, the progress of trials is periodically reported in the console and can also be viewed through integrations such as TensorBoard.

4.3.1. Scaling computation

Each trial in Tune runs in its own Python process, and can be allocated given number of CPU and GPU resources through Ray. Individual trials can themselves leverage distributed computation by launching further subprocesses using Ray APIs. These child processes can

Algorithm	Lines of code
FIFO (trivial scheduler)	10
Asynchronous HyperBand (Li et al. (2018))	78
HyperBand (Li et al. (2016))	215
Median Stopping Rule	68
HyperOpt (Bergstra et al. (2013))	137
Population-Based Training (Jaderberg et al. (2017))	169

Table 1: Model selection algorithms implemented (or integrated) in Tune. Two versions of HyperBand are implemented: the original formulation and the asynchronous variation which is simpler to implement in the distributed setting.

coordinate with each other using Ray to e.g., perform SGD, or use collective communication primitives provided by libraries such as `torch.distributed` and Nvidia NCCL.

4.3.2. Data input

Since each trial in Tune runs as a Ray task or actor, they can use Ray APIs to handle data ingest. For example, weights can be broadcast to all workers using **ray.put(obj)** to the Ray object store, and retrieved via **ray.get(obj_id)** during trial initialization.

5. Implementation

We list in Table 1 currently implemented algorithms in Tune. Line counts include lines used for logging and debugging functionality. We implemented Tune using the Ray (Moritz et al. (2017)) framework, which as noted earlier provides the actor abstraction used to run trials in Tune. In contrast to popular distributed frameworks such as Spark (Zaharia et al. (2012)), or MPI (Gabriel et al. (2004)), Ray offers a more flexible programming model. This flexibility enables Tune's trial schedulers to centrally control the many types of stateful distributed computations created by hyperparameter optimization algorithms.

The Ray framework is also uniquely suited for executing nested computations (i.e., hyperparameter optimization) since it has a *two-level* distributed scheduler. In Ray, task scheduling decisions are typically made on the local machine when possible, only "spilling over" to other machines on the cluster when local resources are exhausted. This avoids any central bottleneck when distributing trial executions that may themselves leverage parallel computations.

6. Conclusion and Future Work

In this work, we explored the design of a general framework for hyperparameter tuning. We proposed the Tune API and System which supports extensible distributed hyperparameter search algorithms while also being easy for end-user model developers to incorporate into their model design processes. We are actively developing new functionality to help not only in the tuning process but also in analyzing and debugging the intermediate results.

References

James Bergstra, Daniel Yamins, and David Daniel Cox. Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures. 2013.

Katharina Eggensperger, Matthias Feurer, Frank Hutter, James Bergstra, Jasper Snoek, Holger Hoos, and Kevin Leyton-Brown. Towards an empirical foundation for assessing bayesian optimization of hyperparameters. 2013.

Matthias Feurer, Aaron Klein, Katharina Eggensperger, Jost Springenberg, Manuel Blum, and Frank Hutter. Efficient and robust automated machine learning. In *Advances in Neural Information Processing Systems*, pages 2962–2970, 2015.

Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary, September 2004.

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. ACM, 2017.

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.

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.

Lisha Li, Kevin Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. Hyperband: A novel bandit-based approach to hyperparameter optimization. *arXiv preprint arXiv:1603.06560*, 2016.

Lisha Li, Kevin Jamieson, Afshin Rostamizadeh, Katya Gonina, Moritz Hardt, Benjamin Recht, and Ameet Talwalkar. Massively parallel hyperparameter tuning, 2018. URL https://openreview.net/forum?id=S1Y7OOlRZ.

Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, William Paul, Michael I Jordan, and Ion Stoica. Ray: A distributed framework for emerging ai applications. *arXiv preprint arXiv:1712.05889*, 2017.

Jasper Snoek, Hugo Larochelle, and Ryan P Adams. Practical bayesian optimization of machine learning algorithms. In *Advances in neural information processing systems*, pages 2951–2959, 2012.

Evan R Sparks, Ameet Talwalkar, Daniel Haas, Michael J Franklin, Michael I Jordan, and Tim Kraska. Automating model search for large scale machine learning. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, pages 368–380. ACM, 2015.

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. ACM, 2013.

Manasi Vartak, Joana M. F. da Trindade, Samuel Madden, and Matei Zaharia. Mistique: A system to store and query model intermediates for. model diagnosis. In *SIGMOD*, 2018.

Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 2–2. USENIX Association, 2012.

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