Enabling Distributed Multi-Objective Bayesian Optimisation using Ax, BoTorch, PyTorch Lightning, RayTune and Custom Scheduling

Sidharrth Nagappan
University of Cambridge
sn666@cam.ac.uk

Abstract

This project implements Multi-Objective Bayesian Optimisation using Ax and BoTorch, using RayTune as a distributed backbone. Notably, it is the first effort to integrate multi-objective BoTorch directly into Ray Tune via the Ax Service API, by modifying RayTune's 'AxSearch' class to allow custom experiments and multiple objectives. It is also the first work to adapt RayTune's existing Hyperband schedulers for multi-objective settings. The empirical results compare critical hyperparameters for these advanced multi-objective schedulers and show that they can reduce runtime by up to 51% while keeping the final hypervolume $\approx 2\%$ of the first-in-first-out (FIFO) scheduling baselines. Hypervolumes and Pareto fronts of dual and triple-objective optimisation settings are also computationally compared and analysed. The complete source code, encompassing custom search algorithms, schedulers and training scripts, is publicly available. [1] [2]

1 Introduction

1.1 Background

In black-box optimisation environments, where objective functions are expensive to evaluate or non-differentiable, Bayesian Optimisation (BO) has emerged as a promising paradigm. These Bayesian methods have been applied to open-ended engineering problems such as hyperparameter tuning, neural architecture search and even drug discovery, where the permutations of possible solutions are infinite, and exhaustive evaluations are unreasonable. Multi-Objective Bayesian Optimisation (MOBO) often extends the scenario to inversely competing objectives, such as maximising accuracy while minimising floating point operations (FLOPs).

With the algorithmic framework built on top of Pareto optimality and scalarization, there are a niche set of libraries, such as BoTorch [2], Optuna [1] and HEBO [3], that implement

[1] https://github.com/sidharrth2002/mobo

[2] Word count excluding abstract is 2376 (computed using *texcount*).

MOBO flavours, each built around their own independent architectures. However, there is limited integration with popular distributed training engines such as Ray, primarily due to the inherent complexity of manipulating the Pareto front in high-dimensional search spaces and the lack of expendable schedulers that can efficiently coordinate budgets for MOBO in distributed settings. This work is the first to patch this gap by:

- re-writing the BoTorch-RayTune API integration to support *distributed* MOBO
- building a *Multi-Objective* Asynchronous Successive Halving (ASHA) scheduler for RayTune based on a naive approximation of the Pareto front. Additionally, beginning exploration of a more sophisticated promotion-driven scheduler based on the algorithm from the MO-ASHA paper [6].

1.2 Novelty

There is a wide range of outstanding GitHub issues on RayTune regarding support for multi-objective optimisation:

- https://github.com/ray-project/ray/issues/8018
- https://github.com/ray-project/ray/issues/32534

Furthermore, *scheduling* for multi-objective optimisation is particularly niche. To the best of my knowledge, this is the first work that explores the possibility of custom RayTune schedulers for multi-objective optimisation. The implementation was extremely time-consuming, especially due to the complex RayTune functional API, and the various sensitive hyperparameters, which work together intricately to control scheduling.

I hope that I will be able to eventually contribute my implementations to the open GitHub issues.

2 Theoretical Framework

2.1 Multi-Objective Bayesian Optimisation

Consider M objectives over a domain X:

$$\min_{\mathbf{x}\in\mathcal{X}} \quad \mathbf{f}(\mathbf{x}) \;=\; \big(f_1(\mathbf{x}), \ldots, f_M(\mathbf{x})\big).$$

Given that each variable has a unique relationship with the objectives, maximising one may interfere with the optimality of another. Therefore, the Pareto front is employed to enumerate non-dominated solutions. A solution is Pareto-optimal if improving one objective would start worsening at least one other objective.

Traditionally, objectives are either modelled independently or as surrogate functions, which are usually Gaussian Processes (GP). Acquisition functions such as Hypervolume Improvement are then used to select new samples that would improve the Pareto front.

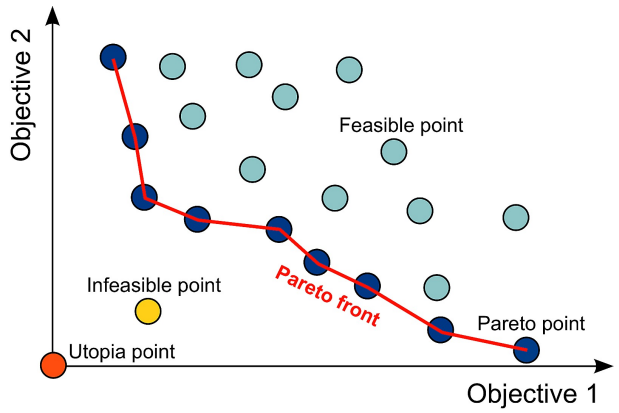

Figure 1: Pareto Front

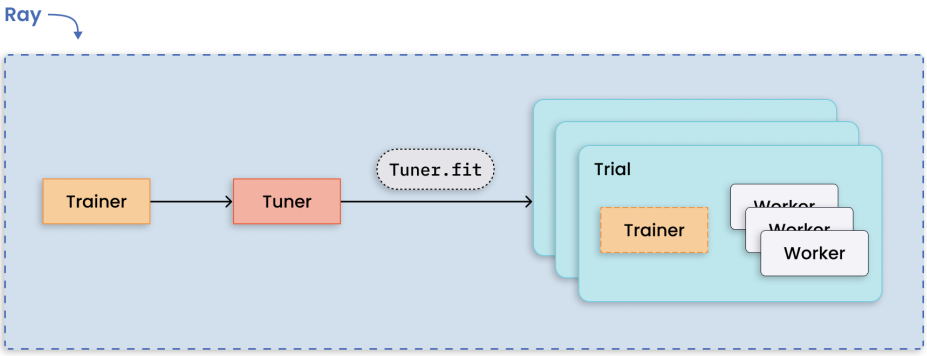

Figure 2: Ray's Distributed Hyperparameter Tuning Orchestration

2.2 Ray and RayTune

Ray is a distributed computing framework that scales single-node experiments across multiple nodes while abstracting away the usual complexity. A Ray Core distributes tasks across nodes in a cluster, using an actor model that allows cross-node communication. Ray spawns worker processes on each node, and efficiently auto-scales runs on massive heterogeneous clusters.

Various auxiliary libraries, including Tune, use the Ray engine, which is particularly designed for *distributed* hyperparameter tuning but broadly supports any search or optimisation flavour (such as neural architecture search). A high-level orchestration of distributed tuning is shown in Figure 2. Bayesian optimisation, as described in detail earlier, is commonly employed when evaluations are computationally expensive and the search space is high-dimensional with multiple local optima. It uses a probabilistic model of an objective function to traverse a search space intelligently, thereby investing resources only in promising directions. To scale BO, Ray integrates with libraries such as Optuna (uses tree-parzen estimators) [1], BoTorch (uses surrogate models) [2] and HEBO (Heteroscedastic Evolutionary Bayesian Optimization) [3].

2.3 Scheduling

Schedulers are a critical component of the Ray stack, responsible for the overall governance of search algorithms. They terminate and pause trials midway based on how promising early performance is. This performance-driven promotion framework ensures that resources are directed to configurations most likely to be successful.

2.3.1 Successive Halving (SHA)

Successive Halving (SH) is a state-of-the-art method that starts experiments with a small resource budget. Rungs are created to level trials based on their promise, with trials either promoted to the subsequent rung (and granted more resources) or terminated. After each rung, SH ranks configurations by performance, discards the worst configurations and promotes the best $1/\eta$ configurations, where η is a reduction factor.

2.3.2 Asynchronous Successive Halving (ASHA)

Asynchronous Successive Halving (ASHA), as shown in Figure 3, adapts SH to distributed settings, by evaluating performance and making a promotion decision as soon as a configuration has exhausted its budget, without the need to synchronise with other running operations. Each configuration would be promoted at most $\log_\eta(R/r_0)$ times, with R being the maximum budget. ASHA is RayTune's most widely used scheduling algorithm. Unlike other tuning frameworks, Ray's take on ASHA focuses on the stopping aspect, such that trials automatically progress to successive rungs if not stopped at some point; essentially, if a trial's performance is below a computed quantile, the trial is immediately and asynchronously stopped. Meanwhile, optimisers like AutoGluon allow the promotion-based paradigm, where decisions are periodically made to promote promising trials to higher resource allocations, rather than focusing primarily on stopping underperforming trials immediately.

2.3.3 Multi-Objective Asynchronous Successive Halving (MO-ASHA)

MO-ASHA extended ASHA to multi-objective contexts, replacing the percentile cutoff, with a Pareto approximation to make promotion decisions [6]. MO-ASHA can either be centred around stopping or promotion. RayTune's Hyperband scheduler adopts the stopping-driven design - a trial will naturally progress to the next rung if not stopped. AutoGluon's implementation makes explicit promotion decisions [4].

Stopping-driven Pareto Front Optimisation When each trial's epoch results are reported, a list of non-dominated past results are computed and a new trial's results are evaluated to see whether they are dominated by any current solutions. A continuation decision is made after allowing a *graceperiod* for trials to warm up, with dominated trials being stopped. Alternatively, scalarisation can be used to compress the multi-objective problem into a single-objective problem, but this is heavily dependent on an efficiently tuned weighting function and has been outperformed by methods that directly operate on the Pareto front.

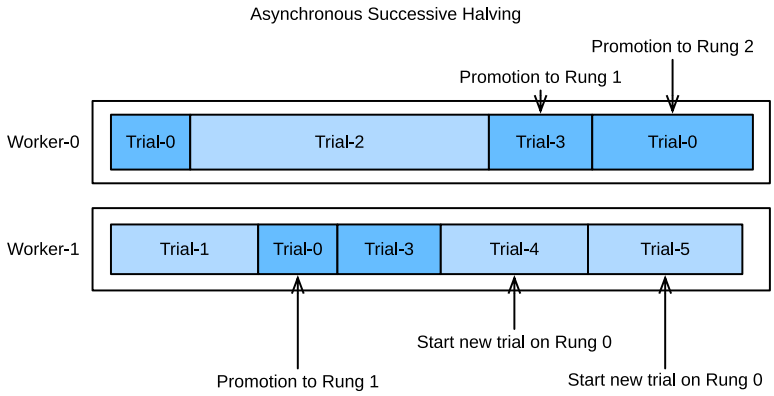

Figure 3: Asynchronous Successive Halving Algorithm

Promotion-driven Pareto Front Optimisation A more sophisticated recent approach makes decisions not just based on whether new trial results are dominated, but also based on whether a new trial would improve the diversity of the Pareto front [6]. It ranks candidates in each front based on the Euclidean distance from past configurations, as a form of non-dominating sort using the ϵ-net (EpsNet) exploration strategy, with the top-ranked candidates being promoted [5].

1. ϵ-net ranks and picks configurations by choosing points that are maximally distant from existing points, to explore the Pareto surface.

2. MO-ASHA uses ϵ-net's selections to promote promising configurations to the next rung.

I implemented the stopping-driven version of MO-ASHA for RayTune by subclassing RayTune's Hyperband scheduler and making individual decisions about when to stop based on whether the trial has already been dominated. I then began working on the promotion-driven version. However, *I later discovered that the promotion-driven approach is incompatible with RayTune's current Hyperband setup because it requires knowledge of the global geometry of the Pareto front. As a result, the ϵ-net rankings could not be used for promotion decisions. Despite this, the initial code for this setup has been made available as a starting point for future research.*

2.4 BoTorch

BoTorch is a state-of-the-art Bayesian Optimisation library, built on top of PyTorch. It supports probabilistic models such as Gaussian Processes (GPs), Deep Kernel Learning (DKL) and other custom variants. It uses differentiable optimization using PyTorch's `autograd` engine and supports multi-objective optimisation. However, BoTorch was primarily designed for research and while its feature suite is incredibly rich, customisation often requires a deep understanding of probabilistic modelling and isn't a regular plug-and-play mechanism. Ax is an initiative to abstract away a lot of BoTorch's complexity, exposing a high-level API for rapid experimentation.

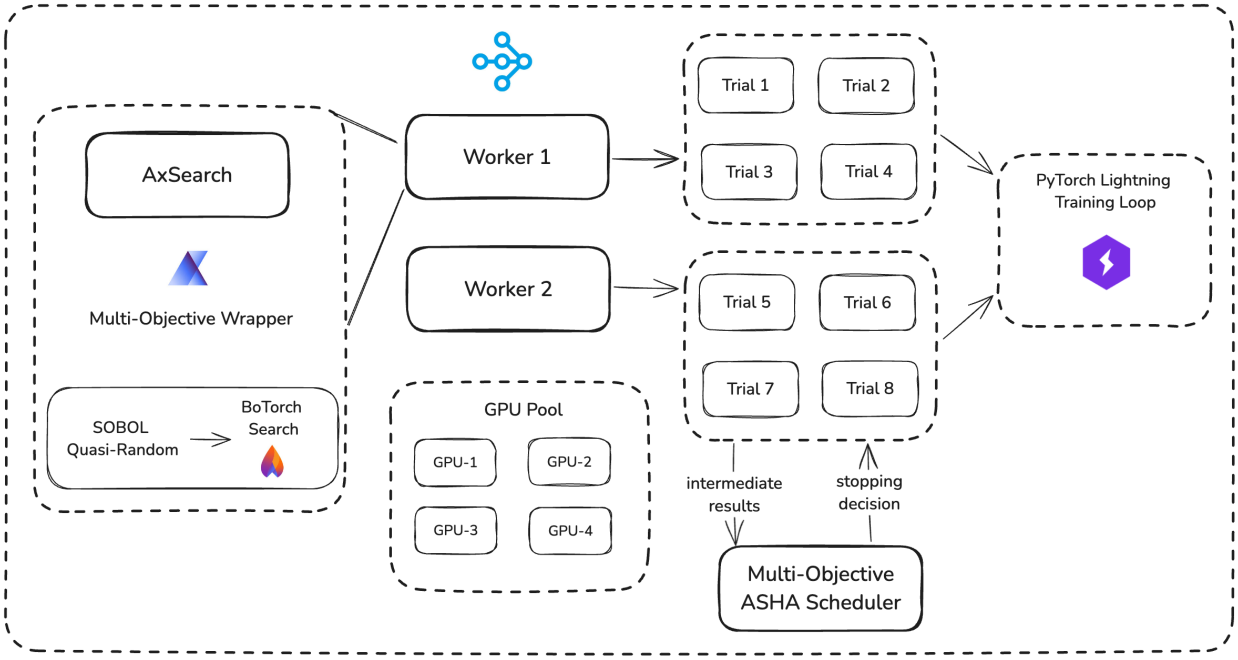

Figure 4: My Full Architecture Employing AxSearch, Multi-Worker Ray, a GPU Pool in the Department of Computer Science's ACS GPU server, a Multi-Objective ASHA scheduler and PyTorch Lightning for training and validation orchestration

Ray does have a rudimentary `AxSearch` class, but it doesn't support custom experiments or multi-objective optimisation. I therefore modify the existing `AxSearch` class to pass custom experiments to Ax's Service API, which in turn speaks to BoTorch.

2.5 Neural Architecture Search (NAS)

Neural Architecture Search (NAS) is a traditional algorithmic problem in deep learning, with the over-arching goal being the discovery of new neural network architectures to solve a given problem. NAS has a large diversity of granularity, from tuning individual layer neurons to choosing macroscopic architecture families (such as the ResNet, MobileNet and Vision Transformer).

NAS can be tackled by a standard random search, grid search reinforcement learning, evolutionary algorithms and BO. BO is a natural solution for NAS because of the expensive cost of building, training and evaluating every possible model variant. It is chosen in this mini-project to demonstrate MOBO.

3 Implementation

To keep the focus on the algorithmic component, NAS is run using a Multi-Layer Perceptron (MLP) on the MNIST dataset. The search space is scoped to the hidden activation neurons of 3 MLP layers, 3 rounds of dropout, the learning rate and batch size; the learning rate is sampled from a logarithmic scale, while the other parameters are categorical. Tuning is

SOBOL → BoTorch With First In First Out Scheduling (FIFO)						
Run	`acc` ↑	`loss` ↓	`params/FLOPs` ↓	Runtime (min.)	Number of Pareto-Optimal Solutions	Hypervolume
8 epochs (exhaustive search)						
1	✓		✓	17m 33s	5	5892.7
2	✓	✓		17m 22s	1	0.028
3		✓	✓	18m 5s	6	32339.78
4	✓	✓	✓	23m 34s	6	2022.97

Table 1: First In First Out (FIFO) Scheduling **Baseline**

done on the University of Cambridge Department of Computer Science's GPU server across 5 GPUs for a maximum of 25 trials; while this setup might be excessive for a small dataset, it allows us to truly explore how (i) distributed data parallelism and (ii) Ray's asynchronous tuning work in tandem. Experiments are scaled across the multi-GPU setup using both Ray's `scaling config` and PyTorch Lightning's distributed training API; 5 are made available to the training script via `CUDA_AVAILABLE_DEVICES` and 3 are chosen, with 1 CPU and 1 GPU being allocated to each worker.

Ax's Service API is called through a custom RayTune `Searcher` class for Multi-Objective Optimisation. When the Ax experiment is instantiated, it decides a progressive search strategy, starting with SOBOL (quasi-random low-discrepancy sequences) to warm up and then BoTorch to go deeper into promising areas.

Runs are orchestrated through a bash script, with `screen` used for runs in the background. Due to heavy activity in the GPU server, there were several CUDA errors during training, with different permutations of GPUs being used to finish running all experiments. However, for comparison fairness, the same number of GPUs was always allocated and the setup was homogeneous.

> *A misconfigured distributed setup ended up in a 23-hour run, that used every GPU in the department's GPU server. I had run it in the background, and I received an email from Malcolm, asking if it was a bug in my code or a big distributed experiment.*

4 Results

The results show that a Pareto-based multi-objective scheduling algorithm can accelerate the search process by up to 51% while producing final configurations whose final performance's disparity is $< 2\%$ of one another. When observing stopping patterns, the scheduler progressively stops at either the 1st, 3rd or final epoch. Furthermore, when comparing the hyperparameters $graceperiod$ and $reductionfactor$ of my scheduler (across Tables 2 and 3), a slightly higher $graceperiod = 2$ produces a richer hypervolume, edging closer to the FIFO baseline.

SOBOL → BoTorch With Stopping-Based MO-ASHA						
Run	`acc` ↑	`loss` ↓	`params/FLOPs` ↓	Runtime (min.)	Number of Pareto-Optimal Solutions	Hypervolume
8 epochs - maxt = 8, graceperiod=1, reductionfactor=6						
1	✓		✓	13m 14s	5	2236.99
2	✓	✓		10m 16s	1	0.008
3		✓	✓	13m 1s	9	27498.52
4	✓	✓	✓	16m 29s	6	905.99

Table 2: Stopping-Based Rudimentary MO-ASHA (maxt = 8, graceperiod=1, reductionfactor=6)

SOBOL → BoTorch With Stopping-Based MO-ASHA						
Run	`acc` ↑	`loss` ↓	`params/FLOPs` ↓	Runtime (min.)	Number of Pareto-Optimal Solutions	Hypervolume
8 epochs - maxt = 8, graceperiod=2, reductionfactor=6						
1	✓		✓	10m 54s	3	4972.79
2	✓	✓		9m 15s	3	0.018
3		✓	✓	11m 14s	3	28510
4	✓	✓	✓	13m 22s	4	1550.7094

Table 3: Stopping-Based Rudimentary MO-ASHA (maxt = 8, graceperiod=2, reductionfactor=6)

SOBOL → BoTorch With Stopping-Based MO-ASHA + ϵ-net Ranking						
Run	`acc` ↑	`loss` ↓	`params/FLOPs` ↓	Runtime (min.)	Number of Pareto Optimal Solutions	Hypervolume
8 epochs - maxt = 8, graceperiod=1, reductionfactor=6						
1	✓		✓	17m 7s	7	5511.22
2	✓	✓		11m 31s	2	0.016
3		✓	✓	13m 30s	4	31474.10
4	✓	✓	✓	23m 14s	10	2074.74

Table 4: Stopping-Based MO-ASHA with additional ϵ-net computation - ϵ-net was not used in the promotion decisions due to an incompatible hyperband design (which I only realised later)

This being said, the MO-ASHA hypervolumes are consistently lower than the FIFO baseline, indicating that the Pareto fronts are of lower quality compared to allowing every trial to reach its terminal state. This discrepancy may arise because trials under MO-ASHA do not complete all epochs. In the current approach, all past trial results are retained in the record space for every Pareto computation, based on the assumption that a trial's new epoch would rightfully dominate its predecessors. This may punish architectures that take several epochs to warm up, but this reasoning is negligible in the case of simple MLPs, as used here.

Notably, the dual minimisation problem of accuracy and loss renders a very limited set of Pareto-optimal solutions. This may be because both parameters are heavily interrelated and often do not compete; upon closer examination, the top configuration for each of these metrics is often the same across the different experimental runs. A narrow spread leads to lower hypervolume. This is also expected because it is not common in the real world to do an accuracy-loss multi-objective optimisation for NAS. It is nevertheless interesting to observe. Across all experimental setups, the triple-objective optimisation problem takes the longest to run and produces the highest number of Pareto-optimal solutions. This is understandable as the trade-offs are more complex, and trial pruning is not as clear-cut.

While ϵ-net was not directly used in promotion decisions, it was used to iteratively analyze the quality of the front. Since my implementation was a manual nested loop (iterating over candidate solutions to estimate diversity), runtimes have slightly increased. However, it is still faster than FIFO scheduling (as seen in Table 4, albeit by a smaller margin. Future work should improve the ϵ-net computation and incorporate it into the promotion decision process.

It must be noted that the time to start training (collecting the data from the centralized store, building the model, instantiating objects in memory) and end training (finishing reporting, saving checkpoints) is longer than the time for the intermediate epochs. Therefore, the benefits of scheduling may be most apparent when the architectures are more complex and training each epoch is more expensive.

Analysing the Pareto fronts gives us insight into the spread of the solutions. As observed in Figure 5, both the *accuracy vs parameters* and *loss vs params* optimisations render a similarly rich distribution across the front, while the non-competing nature of *accuracy vs loss* presents a sub-optimal Pareto front.

5 Conclusion

This work studied Multi-Objective Bayesian Optimisation (MOBO) using BoTorch, powered by an Ax wrapper, Ray's distributed infrastructure, PyTorch Lightning's training orchestration and a custom Pareto-based Multi-Objective Scheduler. The empirical utility of distributed tuning was demonstrated through training on a multi-GPU setup of 5 GPUs on the department's GPU server. The MO-ASHA scheduler was tested with various hyperparameters, and we show that early stopping can produce similar Pareto distributions, and hypervolumes $\approx 16\%$ of one another while improving tuning speed by $\approx 55\%$. Future work can incorporate sophisticated rankings such as ϵ-net to further improve the decisions made by the custom scheduler and minimize the hypervolume gap. While RayTune's Hyperband

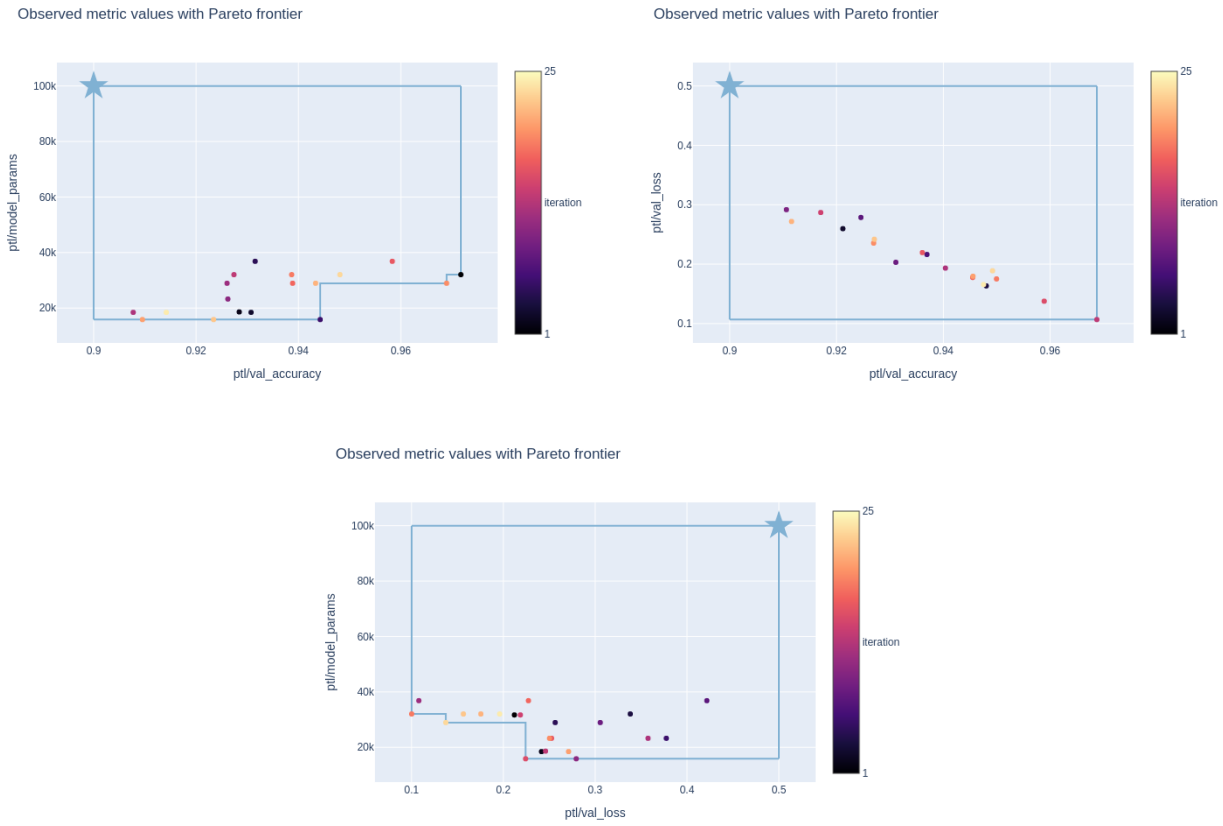

Figure 5: Sample Pareto Fronts after MO-ASHA Scheduling - (i) accuracy and model parameters joint optimisation, (ii) accuracy and loss joint optimisation and (iii) loss and params joint optimisation

scheduler class currently only supports stopping-based schedulers, we hope that this direction will be further explored by future research. Future work can also look into tuning and evaluating scalarization-based multi-objective schedulers.

The work done in this mini-project will be eventually raised as open-source contributions to RayTune's codebase to address open issues related to Ax MOBO and MO-ASHA.

References

[1] Takuya Akiba et al. "Optuna: A Next-generation Hyperparameter Optimization Framework". In: *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. KDD '19. Anchorage, AK, USA: Association for Computing Machinery, 2019, pp. 2623–2631. ISBN: 9781450362016. DOI: 10.1145/3292500.3330701. URL: https://doi.org/10.1145/3292500.3330701.

[2] Maximilian Balandat et al. "BoTorch: Programmable Bayesian Optimization in PyTorch". In: *CoRR* abs/1910.06403 (2019). arXiv: 1910.06403. URL: http://arxiv.org/abs/1910.06403.

[3] Alexander I. Cowen-Rivers et al. "HEBO: Pushing The Limits of Sample-Efficient Hyper-parameter Optimisation". In: *J. Artif. Int. Res.* 74 (Sept. 2022). ISSN: 1076-9757. DOI: 10.1613/jair.1.13643. URL: https://doi.org/10.1613/jair.1.13643.

[4] Nick Erickson et al. "AutoGluon-Tabular: Robust and Accurate AutoML for Structured Data". In: *arXiv preprint arXiv:2003.06505* (2020).

[5] David Salinas et al. "A multi-objective perspective on jointly tuning hardware and hyperparameters". In: *CoRR* abs/2106.05680 (2021). arXiv: 2106.05680. URL: https://arxiv.org/abs/2106.05680.

[6] Robin Schmucker et al. *Multi-objective Asynchronous Successive Halving*. 2021. arXiv: 2106.12639 [stat.ML]. URL: https://arxiv.org/abs/2106.12639.

Appendices

A Code

All implementation code, logs and results are made available on GitHub.