Ray Tune

Distributed hyperparameter optimization made simple

Antoni Baum on behalf of
Anyscale ML team

anyscale

Who am I?

- Software Engineer in Anyscale’s Machine Learning team
- Computer Science & Econometrics MSc Student
- Involved in various open source projects
- linkedin.com/in/yard1/

Agenda

01 Background on Ray and Ray Tune

02 Hyperparameter optimization (HPO) challenges

03 Cutting edge HPO algorithms

04 Distributed HPO

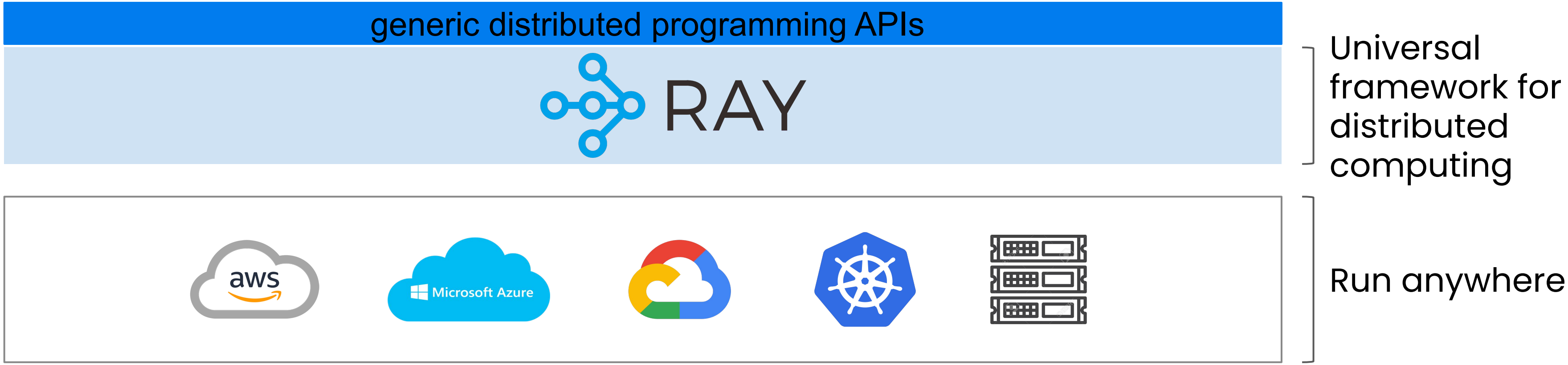
05 Tune-sklearn examples

anyscale

Ray – simple and universal framework for distributed computing

- Berkeley RISELab → Ray (open sourced, python) → Anyscale (commercialize Ray)
- Over 500 users/contributors from a huge number of companies

Ray – simple and universal framework for distributed computing



anyscale

Distributed programming APIs

```python
import ray

# By adding the `@ray.remote` decorator, a regular Python function
# becomes a Ray remote function.
@ray.remote
def my_function():
    return 1

# To invoke this remote function, use the `remote` method.
# This will immediately return an object ref (a future) and then create
# a task that will be executed on a worker process.
obj_ref = my_function.remote()

# The result can be retrieved with ``ray.get``.
assert ray.get(obj_ref) == 1

@ray.remote
def slow_function():
    time.sleep(10)
    return 1

# Invocations of Ray remote functions happen in parallel.
# All computation is performed in the background, driven by Ray's internal event loop.
results = []
for _ in range(4):
    # this doesn't block
    results.append(slow_function.remote())

ray.get(results)
```

Function ➡ Task

Class ➡ Actor

Distributed programming APIs

```python
import ray

# By adding the `@ray.remote` decorator, a regular Python function
# becomes a Ray remote function.
@ray.remote
def my_function():
    return 1

# To invoke this remote function, use the `remote` method.
# This will immediately return an object ref (a future) and then create
# a task that will be executed on a worker process.
obj_ref = my_function.remote()

# The result can be retrieved with ``ray.get``.
assert ray.get(obj_ref) == 1

@ray.remote
def slow_function():
    time.sleep(10)
    return 1

# Invocations of Ray remote functions happen in parallel.
# All computation is performed in the background, driven by Ray's internal event loop.
results = []
for _ in range(4):
    # this doesn't block
    results.append(slow_function.remote())

ray.get(results)
```

Function ➡ Task

Class ➡ Actor

Distributed programming APIs

```python
import ray

# By adding the `@ray.remote` decorator, a regular Python function
# becomes a Ray remote function.
@ray.remote
def my_function():
    return 1

# To invoke this remote function, use the `remote` method.
# This will immediately return an object ref (a future) and then create
# a task that will be executed on a worker process.
obj_ref = my_function.remote()

# The result can be retrieved with ``ray.get``.
assert ray.get(obj_ref) == 1

@ray.remote
def slow_function():
    time.sleep(10)
    return 1

# Invocations of Ray remote functions happen in parallel.
# All computation is performed in the background, driven by Ray's internal event loop.
results = []
for _ in range(4):
    # this doesn't block
    results.append(slow_function.remote())

ray.get(results)
```

Function → Task

Class → Actor

Distributed programming APIs

```python
import ray

# By adding the `@ray.remote` decorator, a regular Python function
# becomes a Ray remote function.
@ray.remote
def my_function():
    return 1

# To invoke this remote function, use the `remote` method.
# This will immediately return an object ref (a future) and then create
# a task that will be executed on a worker process.
obj_ref = my_function.remote()

# The result can be retrieved with ``ray.get``.
assert ray.get(obj_ref) == 1

@ray.remote
def slow_function():
    time.sleep(10)
    return 1

# Invocations of Ray remote functions happen in parallel.
# All computation is performed in the background, driven by Ray's internal event loop.
results = []
for _ in range(4):
    # this doesn't block
    results.append(slow_function.remote())

ray.get(results)
```

Function → Task

Class → Actor

anyscale

Distributed programming APIs

```python
import ray

# By adding the `@ray.remote` decorator, a regular Python function
# becomes a Ray remote function.
@ray.remote
def my_function():
    return 1

# To invoke this remote function, use the `remote` method.
# This will immediately return an object ref (a future) and then create
# a task that will be executed on a worker process.
obj_ref = my_function.remote()

# The result can be retrieved with ``ray.get``.
assert ray.get(obj_ref) == 1

@ray.remote
def slow_function():
    time.sleep(10)
    return 1

# Invocations of Ray remote functions happen in parallel.
# All computation is performed in the background, driven by Ray's internal event loop.
results = []
for _ in range(4):
    # this doesn't block
    results.append(slow_function.remote())

ray.get(results)
```

Function → Task

Class → Actor

anyscale

Distributed programming APIs

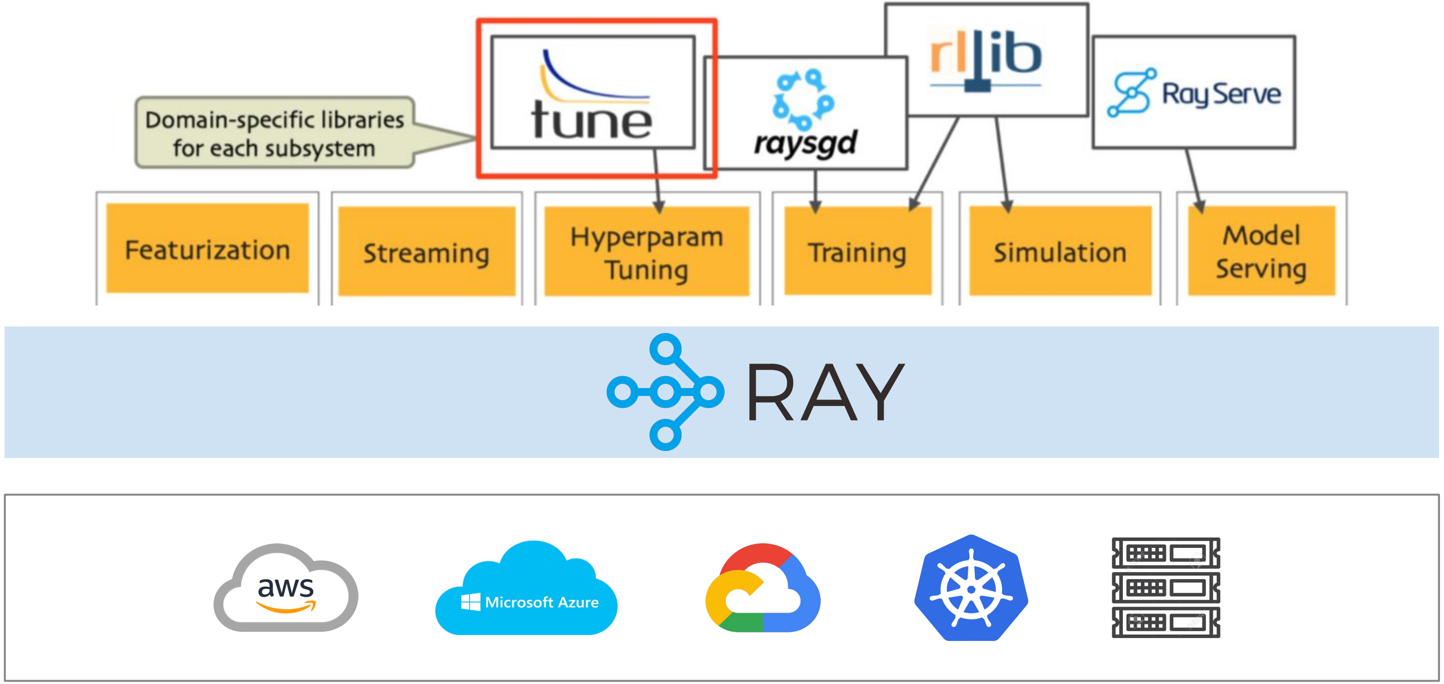
```python
import ray

# By adding the `@ray.remote` decorator, a regular Python function
# becomes a Ray remote function.
@ray.remote
def my_function():
    return 1

# To invoke this remote function, use the `remote` method.
# This will immediately return an object ref (a future) and then create
# a task that will be executed on a worker process.
obj_ref = my_function.remote()

# The result can be retrieved with ``ray.get``.
assert ray.get(obj_ref) == 1

@ray.remote
def slow_function():
    time.sleep(10)
    return 1

# Invocations of Ray remote functions happen in parallel.
# All computation is performed in the background, driven by Ray's internal event loop.
results = []
for _ in range(4):
    # this doesn't block
    results.append(slow_function.remote())

ray.get(results)
```

Function ➡ Task

Class ➡ Actor

anyscale

Capitalize Ray



Library + app ecosystem

Universal framework for distributed computing

Run anywhere

anyscale

Ray Tune - Distributed Hyperparameter Optimization

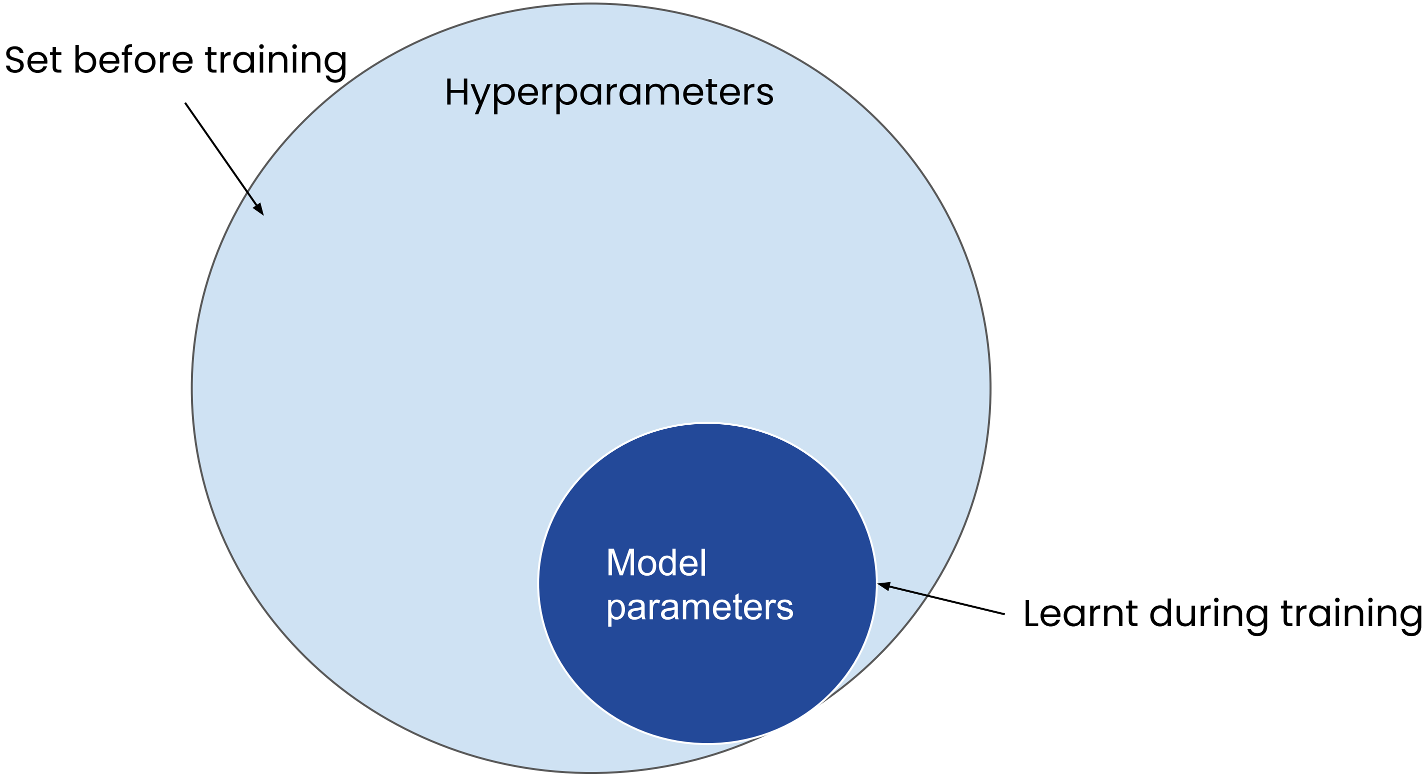
- Provides efficient cutting edge HPO algorithms
- Distributes and coordinates parallel trials in a fault-tolerant and elastic manner
- Saves you time and cost every step of HPO

Technique	Mean Efficiency Gain (%) per Study
Trial-level Early Stopping	16.3
Median Stopping Rule	52.8
Asynchronous Successive Halving Algorithm (ASHA)	68.9
Bayesian Optimization HyperBand (BOHB)	69.9

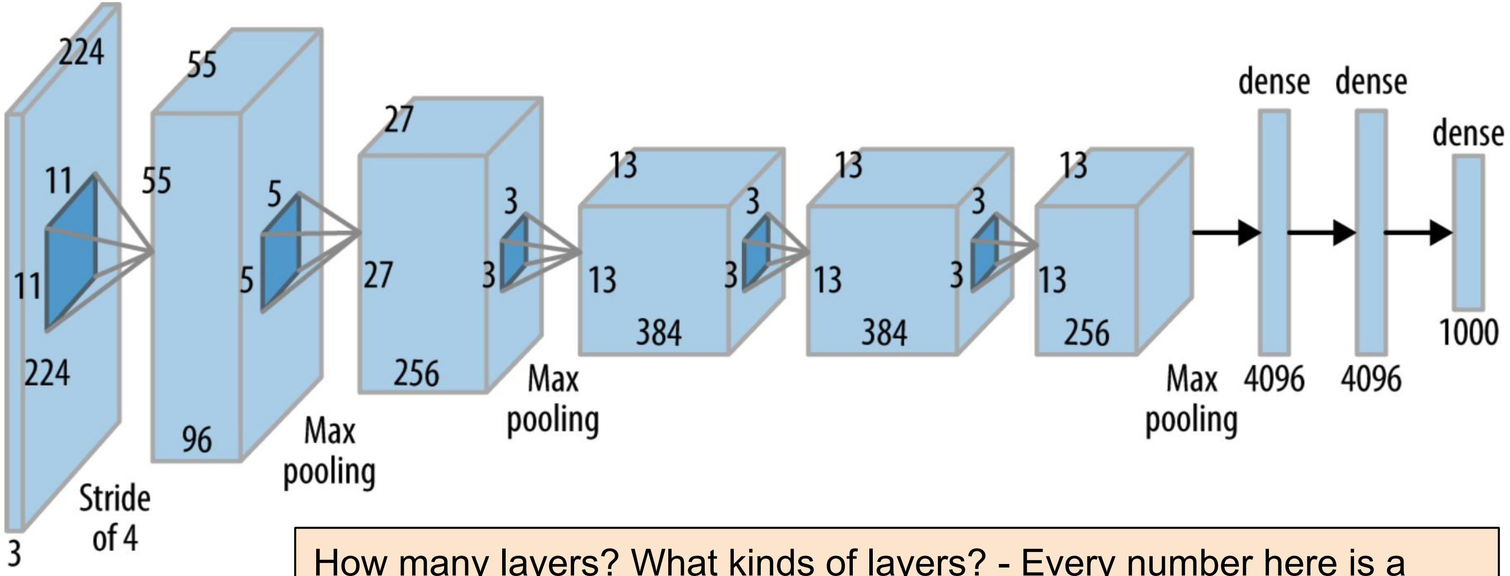
Ray Tune benchmark on 2 weeks of production data at Uber

2X efficiency improvement in terms of GPU/CPU-hours

Hyperparameters

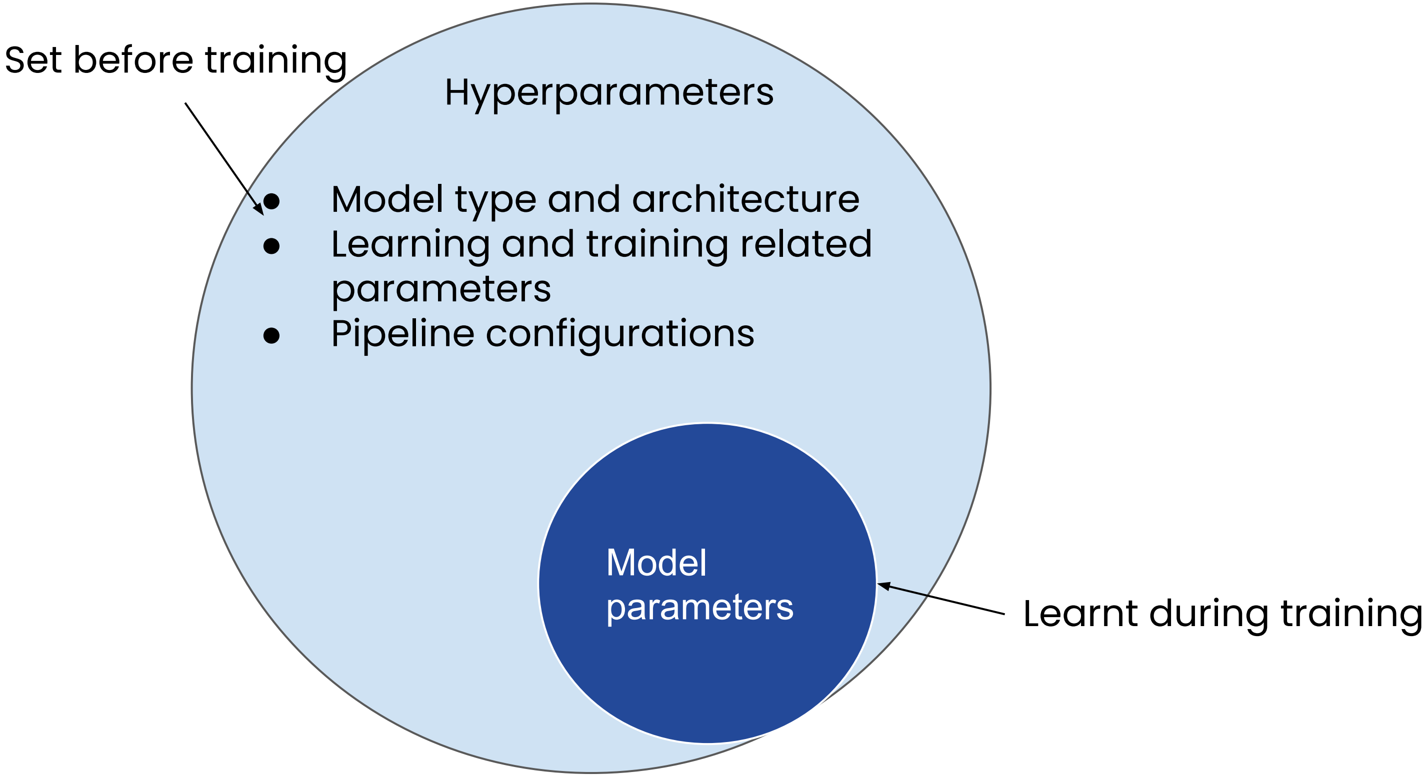


Example



How many layers? What kinds of layers? - Every number here is a hyperparamter!

Hyperparameters



anyscale

Example

Imputer → Categorical encoder → Under/oversampler → XGBoost

Type: Simple or iterative
Simple strategy: Mean or median or constant?

Type: One-hot encoding or label encoding?

Type: SMOTE or random undersampling?
Number of neighbors?

6 - 10 hyperparameters to tune

Total: ~15 hyperparameters to tune!

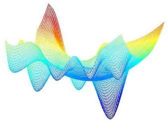
HPO challenges

- Time consuming
  - Large number of combinations
  - Blackbox optimization - the evaluation of each combination (trial) involves model training. Can take days/weeks!
- Resources are expensive (GPUs!)

Ray Tune - distributed HPO

- Efficient algorithms that enable running trials in parallel
- Effective orchestration of distributed trials
- Easy to use APIs

Cutting edge optimization algorithms



Compatible with ML ecosystem

dmlc XGBoost

tune

```
tune.run(train_model)
```

Minimal code changes to work in distributed settings

- Single Process
- Multi-process/ Multi-GPU
- Multi-Node

anyscale

Ray Tune - HPO algorithms

- Over 15 algorithms natively provided or integrated
- Easy to swap out different algorithms with no code change

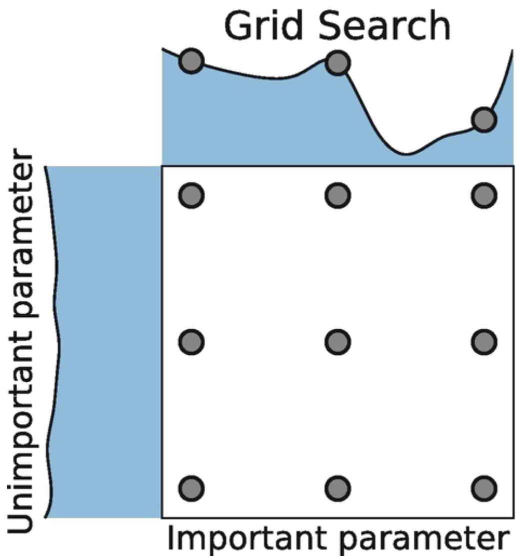
01 **Exhaustive Search**

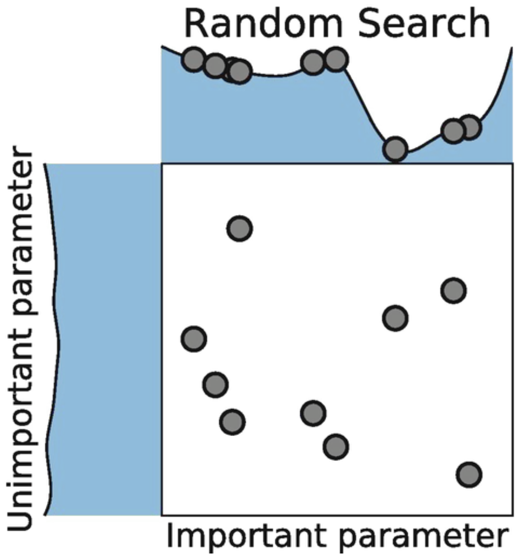
02 **Bayesian Optimization**

03 **Advanced Scheduling**

Exhaustive & Random Search

- Easily parallelizable, easy to implement
- Inefficient, compute intensive

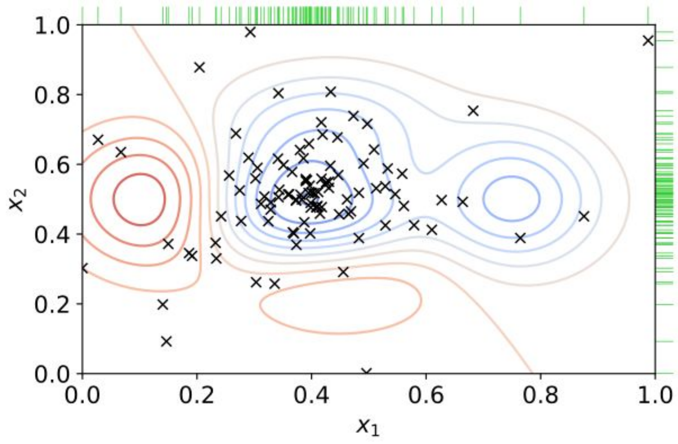




anyscale

Bayesian Optimization

- Uses results from previous combinations (trials) to decide which trial to try next
- Inherently sequential
- Popular libraries:
  - HyperOpt
  - Optuna
  - Scikit-Optimize
  - Nevergrad



https://www.wikiwand.com/en/Hyperparameter_optimization

anyscale

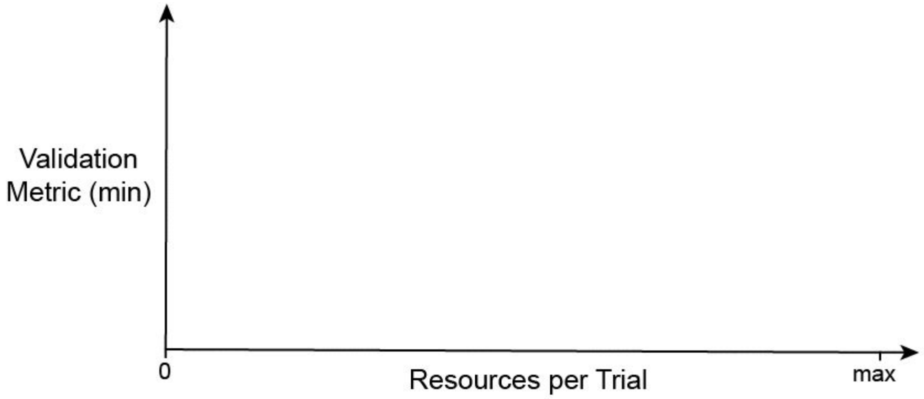
Advanced scheduling

Parallel exploration and exploitation

- Fan out parallel trials during the initial exploration phase
- Make decisions based on intermediate cross-trial evaluations
- Allocate resources to more promising trials
- Early stopping
- Population based training

Advanced Scheduling - Early stopping

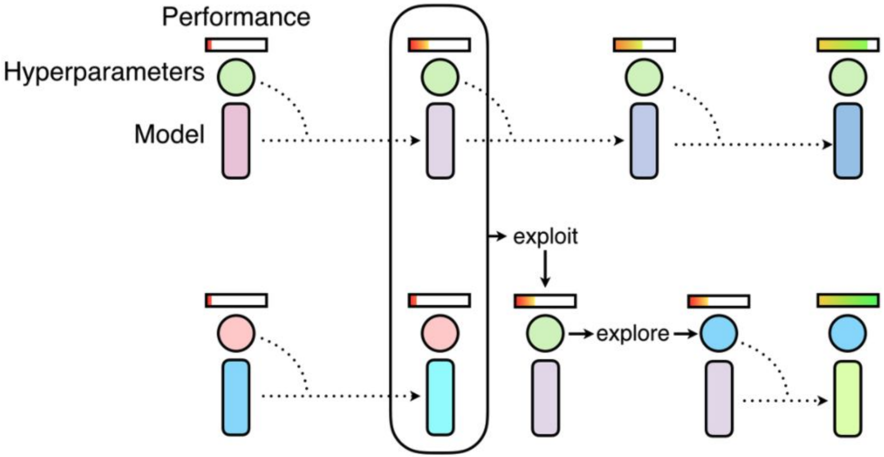
- Fan out parallel trials during the initial exploration phase
- Use intermediate results (epochs, trees, samples) to prune underperforming trials, saving time and computing resources
- Median stopping, Hyperband, ASHA
- Can be combined with Bayesian Optimization (BOHB)



anyscale

Advanced Scheduling - Population Based Training

- Evolutionary algorithm for schedule parameters (eg. learning rate)
- Evaluate a population in parallel
- Terminate lowest performers
- Copy weights of the best performing trials and mutate them



anyscale

Advanced sampling

- **BlendSearch** (by Wang et al.) takes into account the execution time of combinations, progressively trying more expensive ones. It combines local and global search.
- **Heteroscedastic Evolutionary Bayesian Optimisation** (by Cowen-Rivers, Lyu et al.) combines BO with evolutionary algorithms. Winner of the NeurIPS 2020 black-box optimisation competition.
- **BOHB** (Falkner et al.) combines BO with HyperBand, making informed decisions based on partial results.

Woohoo!

Let’s review what we have talked about.

- There are various HPO algorithms with a trend of going parallel
- More advanced ones are often hard to implement
    - Even more so in a distributed setting
- All of the different libraries implementing HPO algorithms have different APIs and functionality

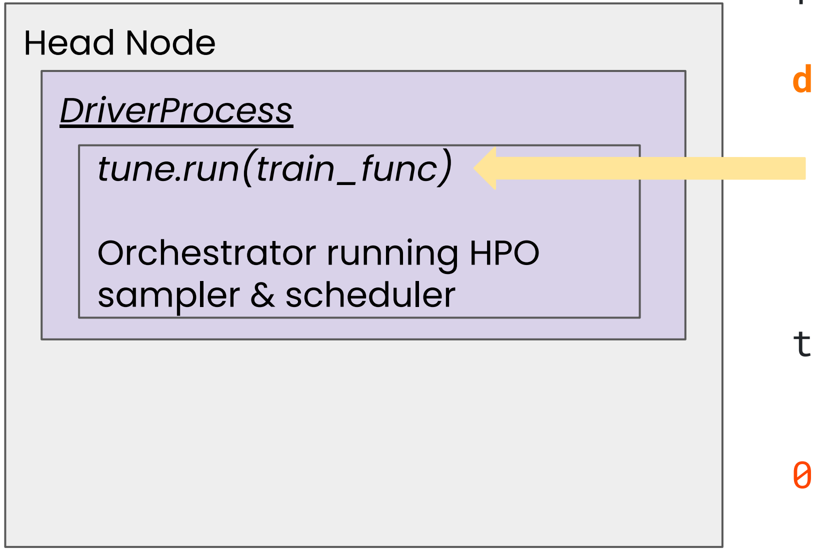
Good news!

- Ray Tune implements and integrates with all these algorithms and their parent libraries
- Bring your own algorithms!
- Allows user to swap out different algorithms very easily and take benefit from a unified API

Architecture requirements

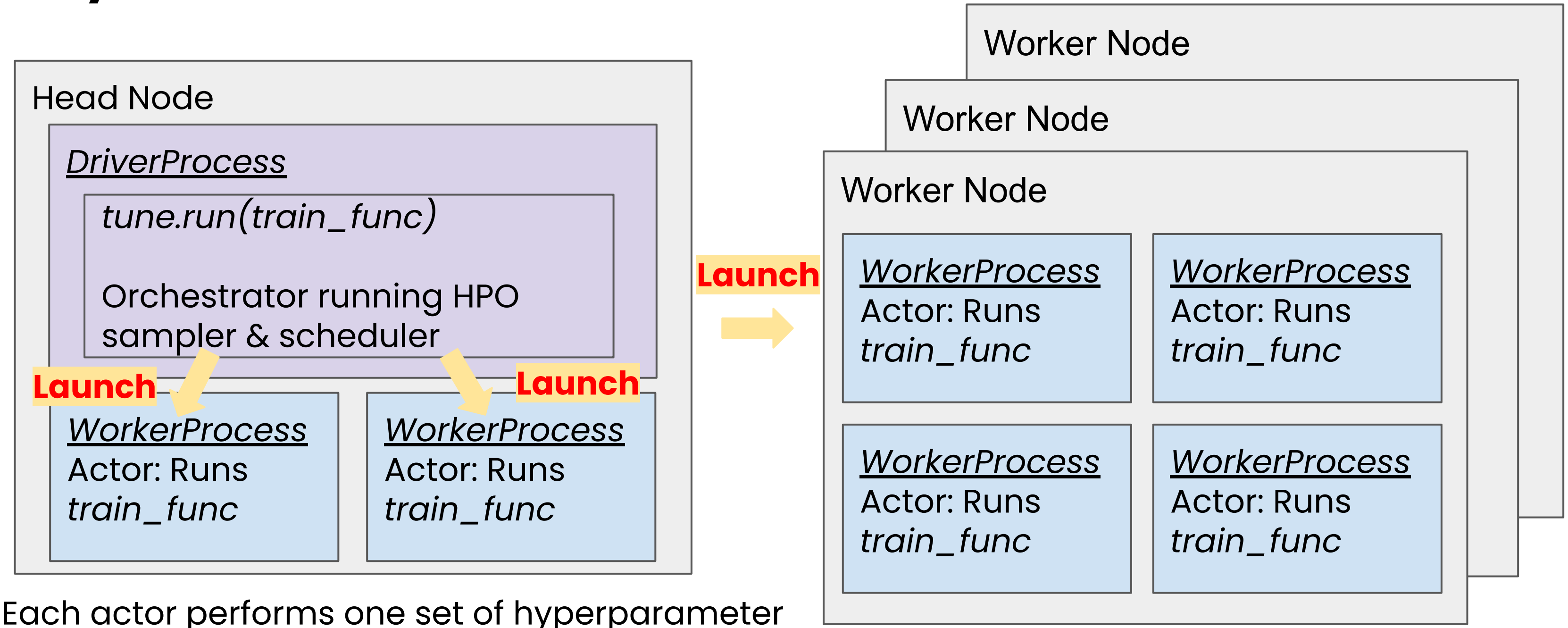
- Distributed HPO imposes a set of architecture requirements
- Granular control over when to start, pause, early stop, restore, or mutate each trial at specific iterations with little overhead
- Master-worker architecture that centralizes decision making
  - Sampler - providing combinations to evaluate
  - Scheduler - which trials to start, pause and stop
- Elasticity and fault tolerance

Ray Tune - *distributed* HPO



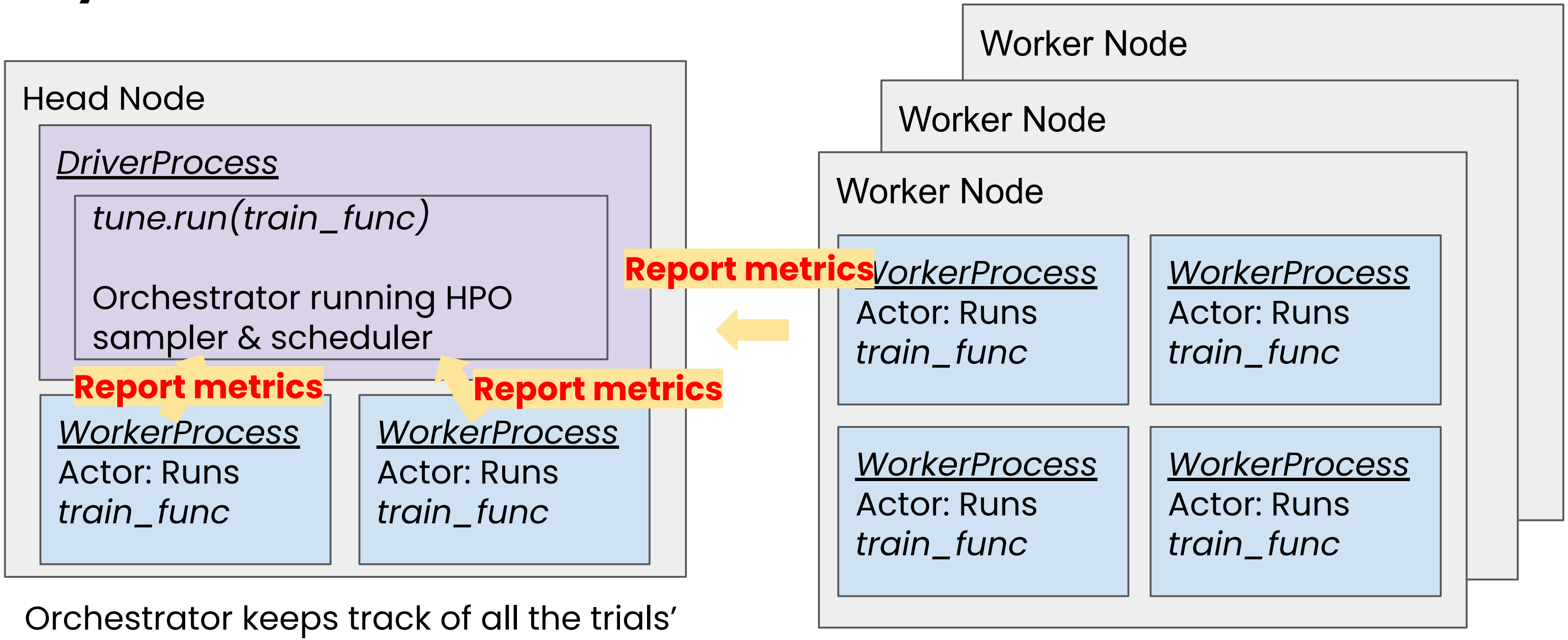

```
from ray import tune

def train_func(config):
    model = ConvNet(config)
    for i in range(epochs):
        current_loss = model.train()
        tune.report(loss=current_loss)

tune.run(
    train_func,
    config={“alpha”: tune.uniform(0.001, 0.1)},
    num_samples=100,
    scheduler=“asha”,
    search_alg=”optuna”)
```

Ray Tune - *distributed* HPO



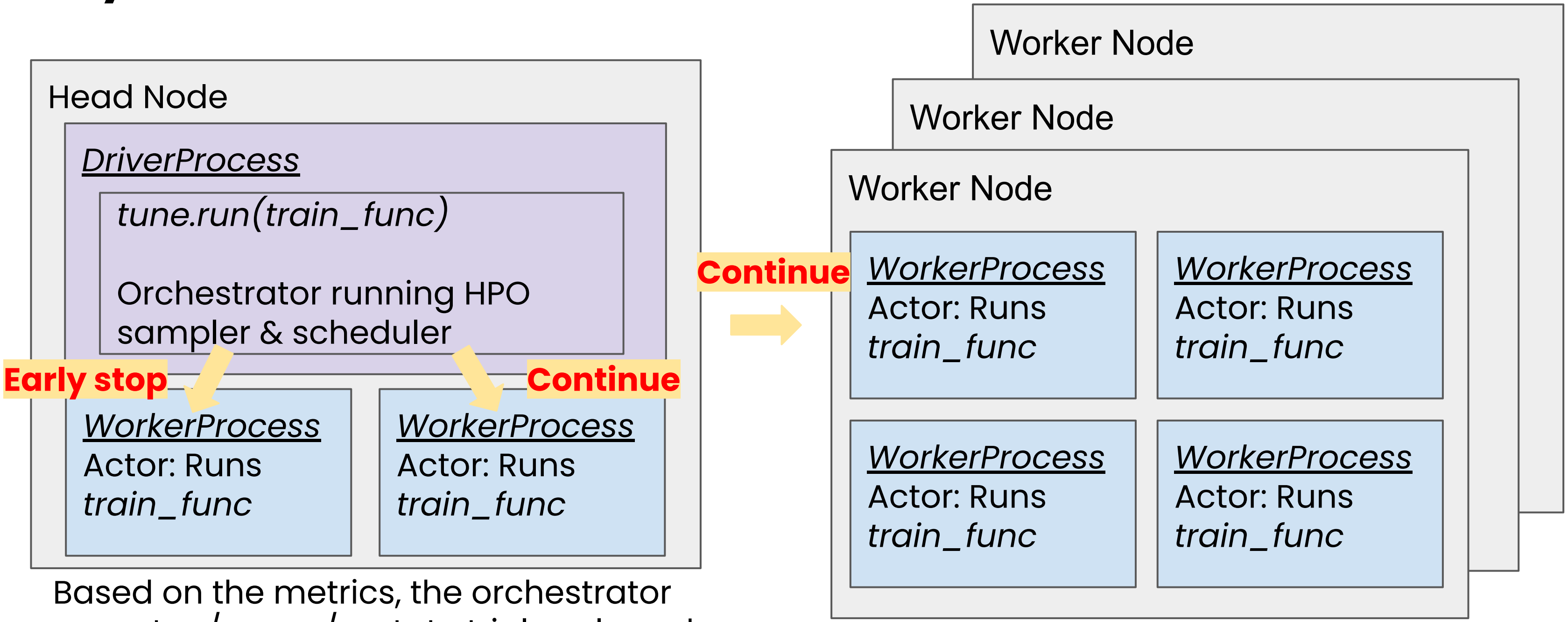
Each actor performs one set of hyperparameter combination evaluation (a trial)

Ray Tune - *distributed* HPO



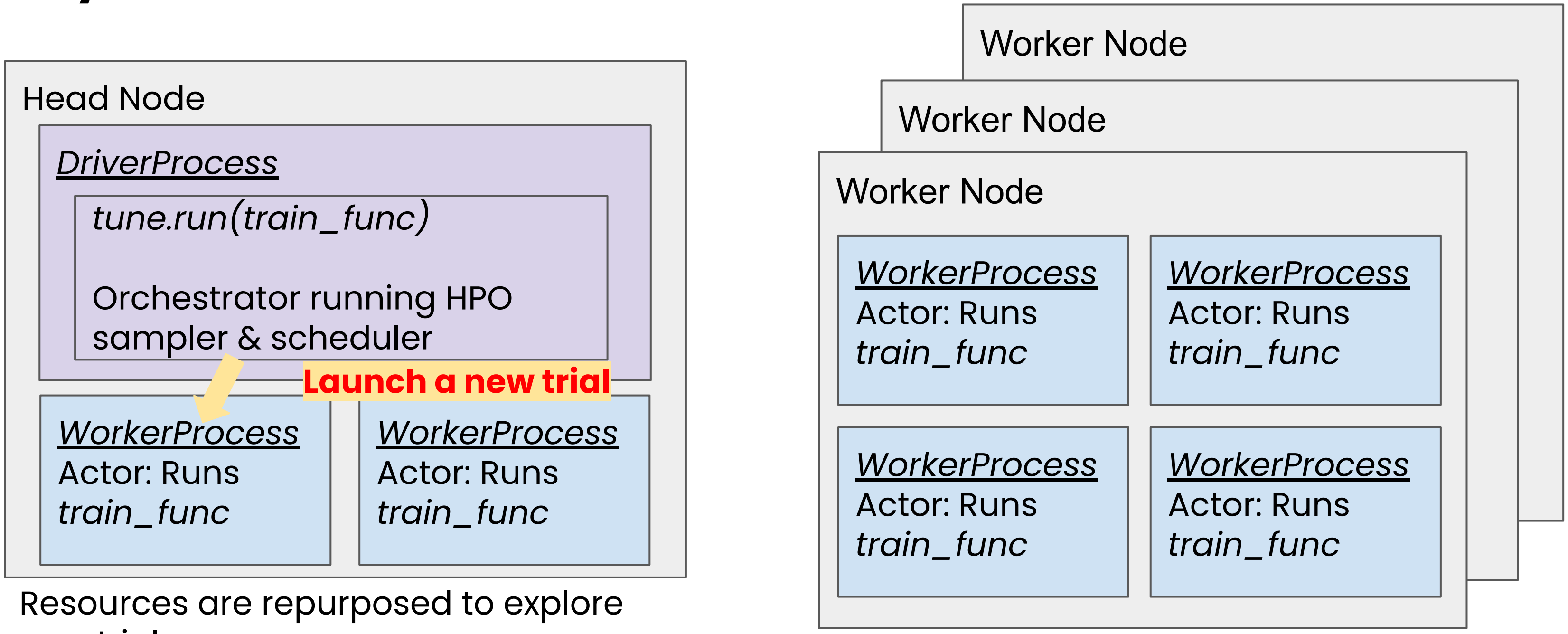
Orchestrator keeps track of all the trials' progress and metrics.

Ray Tune - *distributed* HPO



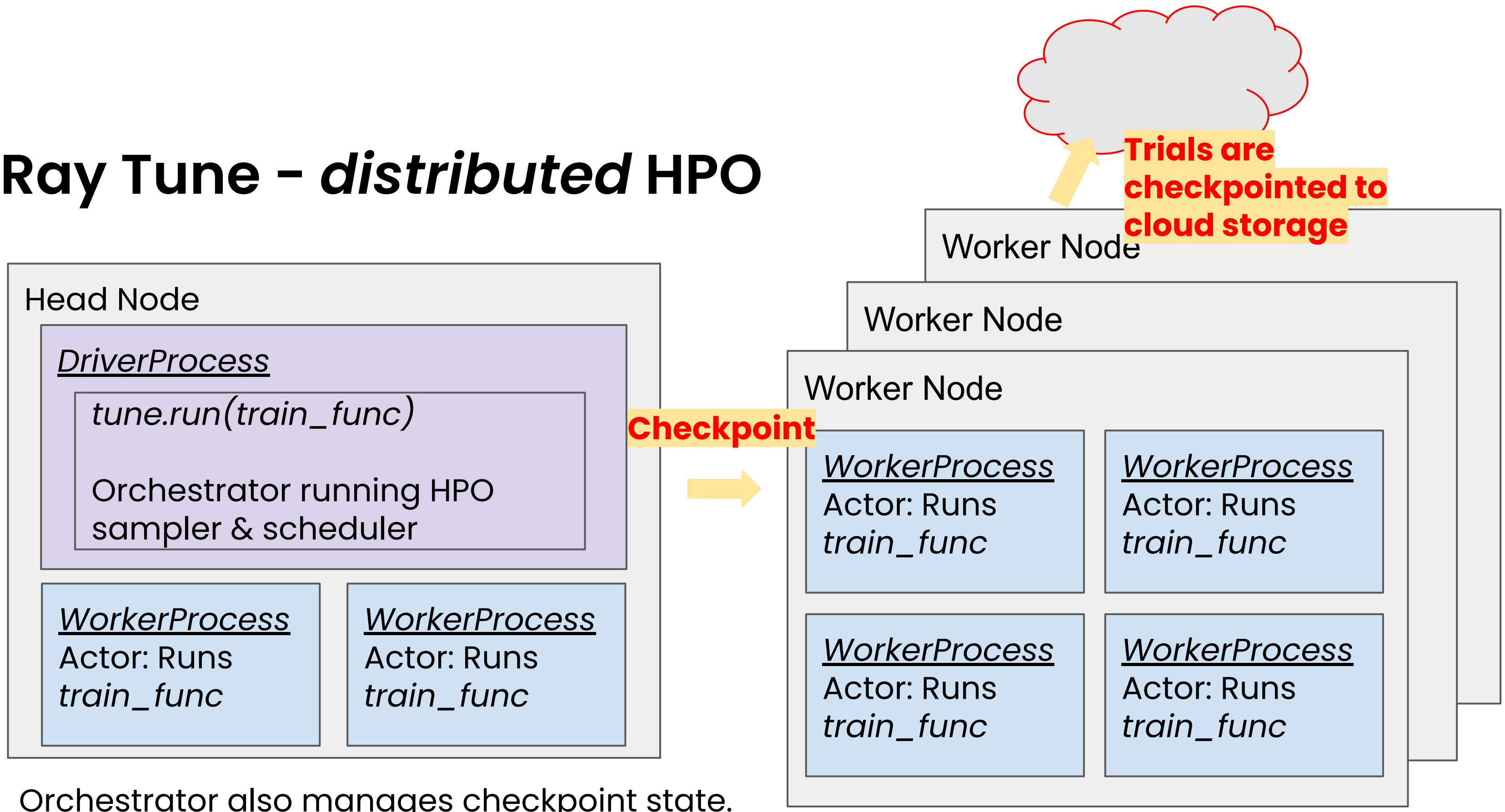
Based on the metrics, the orchestrator may stop/pause/mutate trials or launch new trials when resources are available.

Ray Tune - *distributed* HPO



Resources are repurposed to explore new trials.

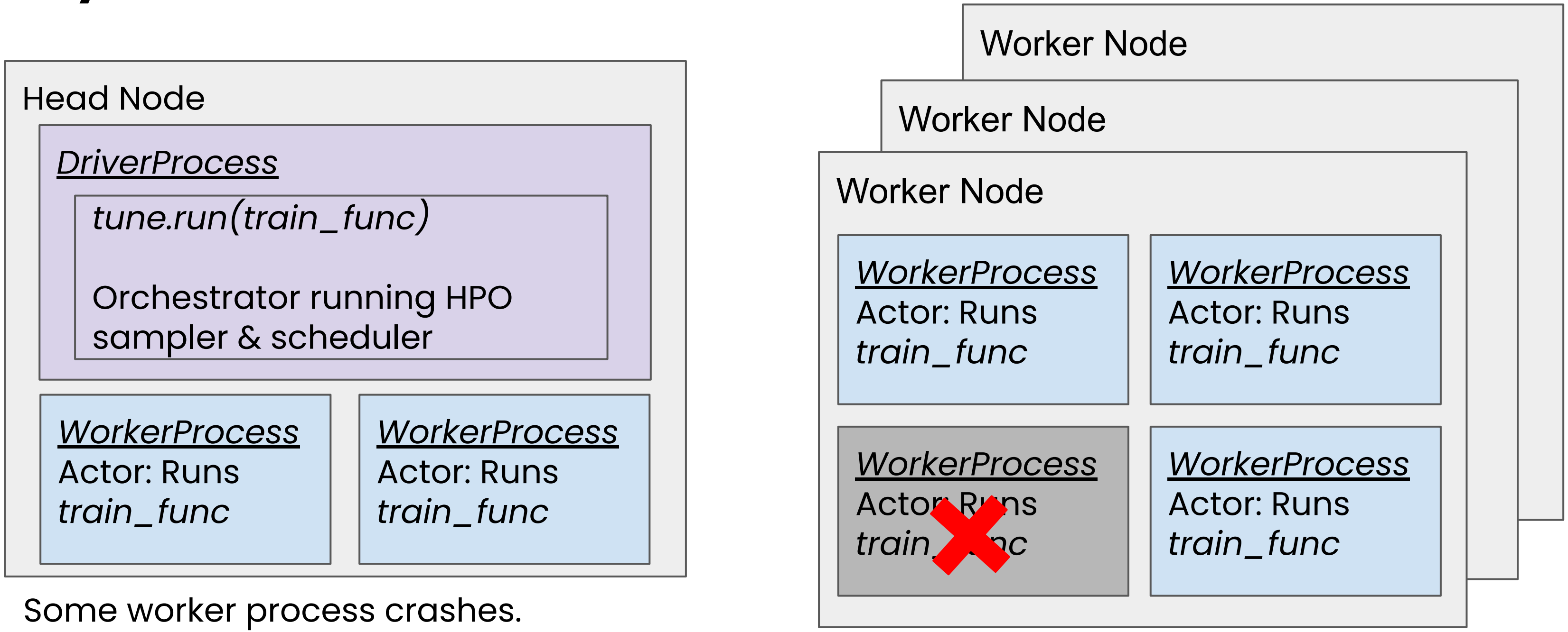
Ray Tune - *distributed* HPO



Orchestrator also manages checkpoint state.

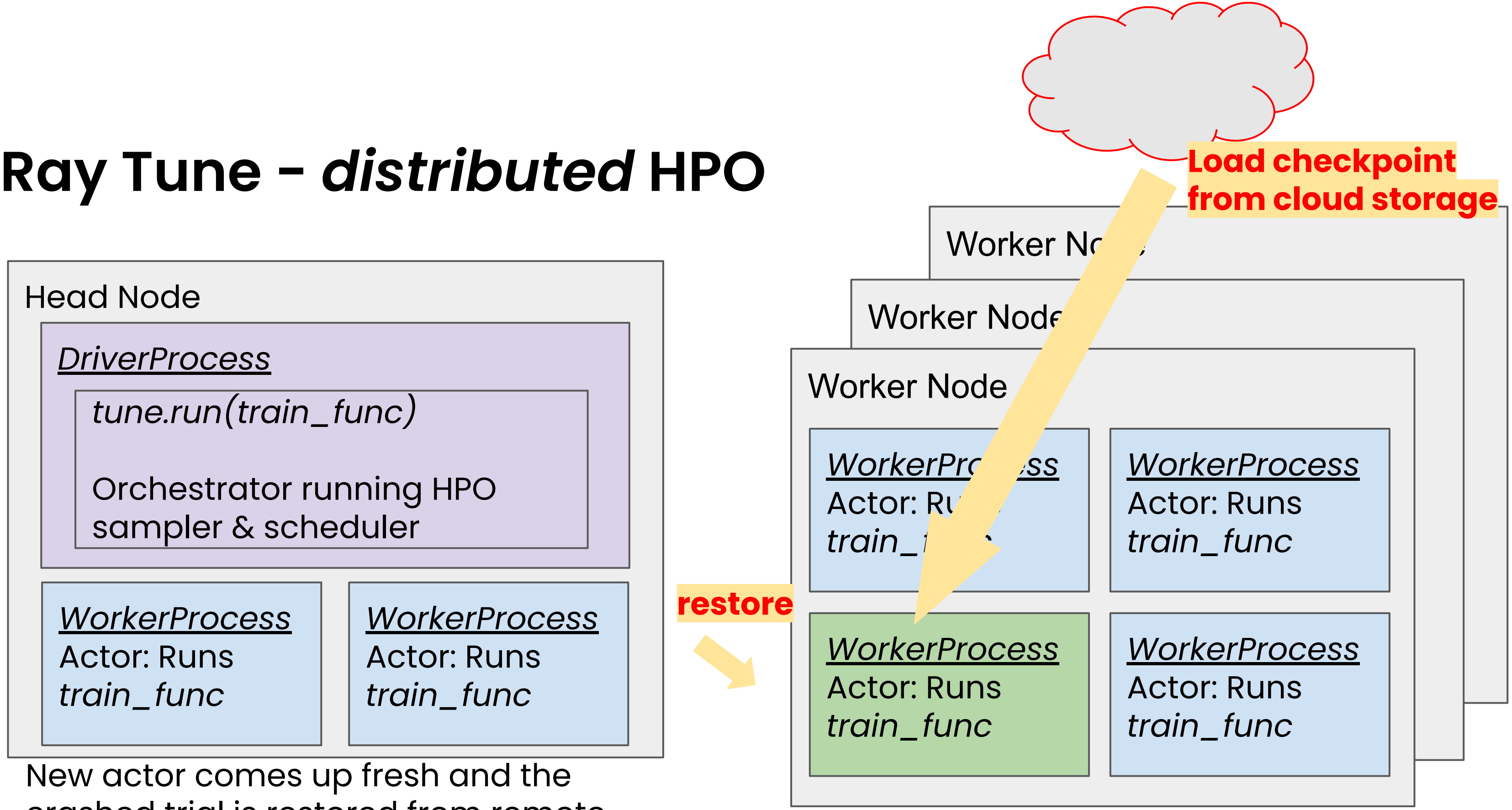
anyscale

Ray Tune - *distributed* HPO



Some worker process crashes.

Ray Tune - *distributed* HPO



New actor comes up fresh and the crashed trial is restored from remote checkpoint.

anyscale

Woohoo!

Let's review what we have talked about.

What makes Ray Tune special

- Provides efficient HPO algorithms
- Distributes and coordinates parallel trials in a fault-tolerant and elastic manner
- Integrated with ML ecosystem

Tune-sklearn

- A scikit-learn wrapper for Ray Tune
  - drop-in replacement for scikit-learn model selection module (RandomizedSearchCV and GridSearchCV)
- Provides a familiar and simple API for advanced, distributed HPO
- https://github.com/ray-project/tune-sklearn

Drop-in replacement

```
from sklearn.model_selection import GridSearchCV

parameters = {
    'alpha': [1e-4, 1e-1, 1],
    'epsilon':[0.01, 0.1]
}
search = GridSearchCV(
    SGDClassifier(),
    parameters,
    n_jobs=-1
)

search.fit(X_train, y_train)
```

Use all the cores on the single machine

anyscale

Drop-in replacement

```python
from tune_sklearn import TuneSearchCV

parameters = {
    'alpha': [1e-4, 1e-1, 1],
    'epsilon':[0.01, 0.1]
}
search = TuneSearchCV(
    SGDClassifier(),
    parameters,
    n_jobs=-1
)

search.fit(X_train, y_train)
```

Use all the resources throughout the entire cluster!

tune-sklearn demo

- Driver safety prediction
- Cluster started through Ray cluster launcher
- 5*8 CPUs
- Jupyter notebook that runs on head node

Thank You

Let’s keep in touch!

- https://ray.io/
- https://discuss.ray.io/
- Ray slack
- https://github.com/ray-project/tune-sklearn

anyscale

Q & A