Scaling AI with Ray

pub.towardsai.net/scaling-ai-with-ray-faeb5434c2b6

Luhui Hu February 11, 2023

Ray is emerging in AI engineering and becomes essential to scale LLM and RL

Photo by on

Spark is almost essential in data engineering. And Ray is emerging in AI engineering.

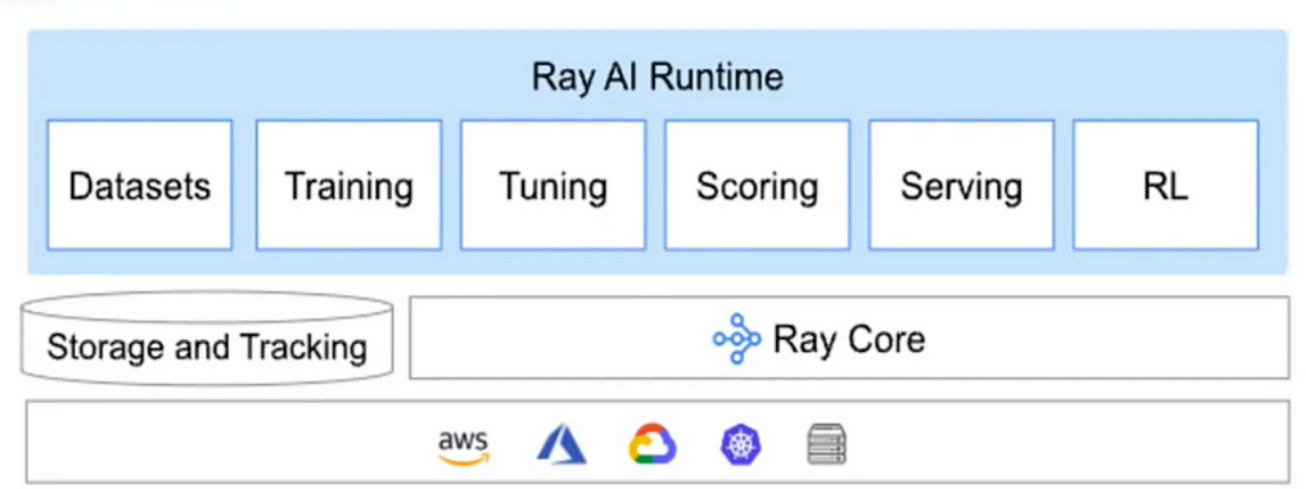
Ray is a successor to Spark from UCB. Spark and Ray have many similarities, e.g., unified engines for computing. But Spark is mainly focused on large-scale data analytics, while Ray is designed for machine learning applications.

Here, I'll introduce Ray and touch on how to scale large language models (LLM) and reinforcement learning (RL) with Ray, then wrap up with Ray's nostalgia and trend.

Introduction to Ray

Ray is an open-source unified compute framework making it easy to scale AI and Python workloads, from reinforcement learning to deep learning to model tuning and serving.

Below is Ray's latest architecture. It mainly has three components: Ray Core, Ray AI Runtime, and Storage and Tracking.



Ray 2.x and Ray AI Runtime (AIR) (Source: January 2023)

Ray Core provides a small number of core primitives (i.e., tasks, actors, objects) for building and scaling distributed applications.

Ray AI Runtime (AIR) is a scalable and unified toolkit for ML applications. AIR enables simple scaling of individual workloads, end-to-end workflows, and popular ecosystem frameworks, all in just Python.

AIR builds on Ray's best-in-class libraries for Preprocessing, Training, Tuning, Scoring, Serving, and Reinforcement Learning to bring together an ecosystem of integrations.

Ray enables seamless scaling of workloads from a laptop to a large cluster. A Ray cluster consists of a single head node and any number of connected worker nodes. The number of worker nodes may be *autoscaled* with application demand as specified by Ray cluster configuration. The head node runs the autoscaler.

We can submit jobs for execution on the Ray cluster or interactively use the cluster by connecting to the head node and running `ray.init`.

It's simple to start and run Ray. The following will illustrate how to install it.

```
$ pip install ray[████████████████████████████████████████] 100%Successfully installed
ray$ python>>>import ray; ray.init() ... INFO worker.py:1509 -- Started a  Ray
instance. View the dashboard at 127.0.0.1:8265 ...

pip install -U  pip install -U   pip install -U   pip install -U
```

Furthermore, Ray can run at scale on Kubernetes and cloud VMs.

Scale LLM and RL with Ray

ChatGPT is a significant AI milestone with rapid growth and unprecedented impact. It is built on OpenAI's GPT-3 family of large language models (LLM) employing Ray.

Greg Brockman, CTO and cofounder of OpenAI, said, *At OpenAI, we are tackling some of the world's most complex and demanding computational problems. Ray powers our solutions to the thorniest of these problems and allows us to iterate at scale much faster than we could before.*

It takes about 25 days to train GPT-3 on 240 ml.p4d.24xlarge instances of the SageMaker training platform. The challenge is not just processing but also memory. Wu Tao 2.0 appears to need more than 1000 GPUs only to store its parameters.

Training ChatGPT, including large language models like GPT-3 requires substantial computational resources and is estimated to be in the tens of millions of dollars. By empowering ChatGPT, we can see the scalability of Ray.

Ray tries to tackle challenging ML problems. It supports training and serving reinforcement learning models from the beginning.

Let's code in Python to see how to train a large-scale reinforcement learning model and serve it using Ray Serve.

Step 1: Install dependencies for reinforcement learning policy models.

```
!pip install -qU  gym
```

Step 2: Define training, serving, evaluating, and querying a large-scale reinforcement learning policy model.

```
import gym
import numpy as np
import requests

# import Ray-related libs
from ray.air.checkpoint import Checkpoint
from ray.air.config import RunConfig
from ray.train.rl.rl_trainer import RLTrainer
from ray.air.config import ScalingConfig
from ray.train.rl.rl_predictor import RLPredictor
from ray.air.result import Result
from ray.serve import PredictorDeployment
from ray import serve
from ray.tune.tuner import Tuner

# train API for RL by specifying num_workers and use_gpu
deftrain_rl_ppo_online(num_workers: , use_gpu:  = ) -> Result:
print("Starting online training")
    trainer = RLTrainer(
        run_config=RunConfig(stop={"training_iteration": 5}),
        scaling_config=ScalingConfig(num_workers=num_workers, use_gpu=use_gpu),
        algorithm="PPO",
        config={
"env": "CartPole-v1",
"framework": "tf",
        },
    )



    tuner = Tuner(
        trainer,
        _tuner_kwargs={"checkpoint_at_end": True},
    )
    result = tuner.fit()[0]
return result



# serve RL model
defserve_rl_model(checkpoint: Checkpoint, name=) -> str:

""" Serve an RL model and return deployment URI.

    This function will start Ray Serve and deploy a model wrapper    that loads the
RL checkpoint into an RLPredictor.    """
    serve.run(
        PredictorDeployment.options(name=name).bind(
            RLPredictor, checkpoint
        )
    )
returnf"http://localhost:8000/"
```

```
# evaluate RL policy
defevaluate_served_policy(endpoint_uri: , num_episodes:  = ) -> list:

""" Evaluate a served RL policy on a local environment.

    This function will create an RL environment and step through it.    To obtain the
actions, it will query the deployed RL model.    """
    env = gym.make("CartPole-v1")



    rewards = []
for i inrange(num_episodes):
        obs = env.reset()
        reward = 0.0
        done = False
whilenot done:
            action = query_action(endpoint_uri, obs)
            obs, r, done, _ = env.step(action)
            reward += r
        rewards.append(reward)



return rewards

 ():

""" Perform inference on a served RL model.

    This will send an HTTP request to the Ray Serve endpoint of the served    RL
policy model and return the result.    """    action_dict =
requests.post(endpoint_uri, json={: obs.tolist()}).json()     action_dict
```

Step 3: Now train the model, serve it using Ray Serve, evaluate the served model, and finally shut down Ray Serve.

```
# training in 20 workers using GPU
result = train_rl_ppo_online(num_workers=20, use_gpu=True)

# serving
endpoint_uri = serve_rl_model(result.checkpoint)

# evaluating
rewards = evaluate_served_policy(endpoint_uri=endpoint_uri)

serve.shutdown()
```

Ray Nostalgia and Trend

Ray was initiated as a research project at RISELab of UCB. RISELab is the successor of AMPLab, where Spark was born.

Professor Ion Stoica is the soul of Spark and Ray. He initiated to found Databricks with Spark and Anyscale with Ray as their core products.

I was privileged to work with RISELab fellows in its early stage and witnessed Ray come into being.

Ray: A Distributed Execution Framework for Emerging AI Applications

Contributors: Michael I. Jordan, Richard Liaw, Philipp Moritz, Mehrdad Niknami, Robert Nishihara, William Paul, Johann Schleier-Smith, Ion Stoica, Alexey Tumanov, Stephanie Wang

University of California, Berkeley

Why Build a New System?

Machine learning applications are changing from *static* to *dynamic*.

Question answering systems → **Dialogue systems**
Medical diagnosis → **Real-time health recommendations**
Course recommendations → **Intelligent tutoring systems**

The changing nature of ML applications imposes new requirements on the systems we use.

- Support **heterogeneous tasks** (simulations, training, search, deep learning).
- Support **small tasks, high throughput** (need to do lots of simulations in parallel).
- Must be **low latency** (RL is reactive/real-time in nature).
- Allow **arbitrary fine-grained task dependencies** (multiple data streams, complex algorithms/patterns). Bulk synchronous parallel is not enough.

The Ray Programming Model

- `ray.remote` defines a *remote function* or a *remote actor*.
- `ray.get` fetches the values of remote futures.
- `ray.wait` waits for a subset of futures to be available.

A Reinforcement Learning Experiment

Highlighted code shows the changes needed to parallelize the code using Ray.

```
@ray.remote
def experiment(config):
  policy = new_policy(config)
  for _ in range(100):
    # Launch 200 simulations in parallel and aggregate the results.
    gradients = ray.get([simulation.remote(policy, seed=i)
                         for i in range(100)])
    # Update the model.
    policy.update(gradients)
  return policy

# Launch a number of experiments in parallel.
ray.get([experiment.remote(config) for config in config_list])
```

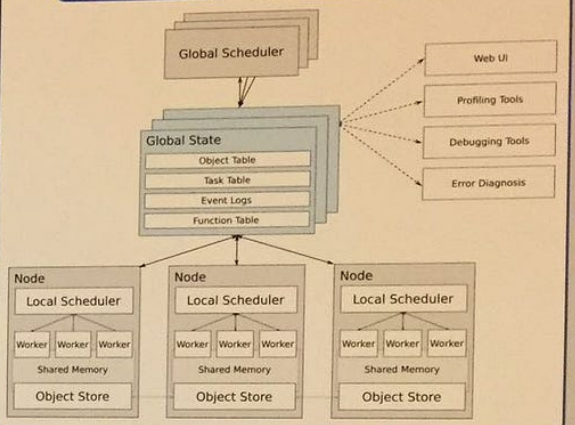
Weight-Averaging Networks

```
@ray.remote
class Actor(object):
  def __init__(self):
    self.network = ConvNet()

  def step(self, weights, images, labels):
    # Train network for NUM_STEPS subsampling from images and labels.
    self.network.set_weights(weights)
    self.network.step(images, labels, steps=NUM_STEPS)
    return self.network.get_weights()

  def get_weights(self):
    return self.network.get_weights()

actors = [Actor.remote() for _ in range(NUM_ACTORS)]
weights = actors.get_weights.remote()
for _ in range(NUM_ITERS):
  weights_to_average = []
  for actor in actors:
    # Get a random batch of images and labels to subsample from.
    images, labels = get_random_batch()
    weights_to_average.append(actor.step.remote(weights, images, labels))
  average_weights = np.mean(ray.get(weights_to_average))
  weights = ray.put(average_weights)
```

Related Systems

- **MPI:** No scheduling, no fault tolerance. Difficult to do adaptive computation.
- **Spark:** Bulk synchronous parallel execution model versus a fine-grained task-parallel model.
- **TensorFlow/MxNet:** Operating at different levels of abstraction, we provide more flexibility in enabling custom operations and scheduling tasks dynamically and automatically.
- **Dask:** Our architecture enables much higher performance and fuses futures and actors into the same dataflow graph.

Architecture

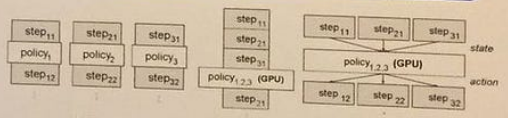


Evolution Strategies

	10 nodes	20 nodes	30 nodes	40 nodes	50 nodes
Reference (steps/sec)	97K	215K	202K	N/A	N/A
Ray (steps/sec)	152K	285K	323K	476K	571K

- A Ray implementation that uses evolutionary algorithms in reinforcement learning.

Rollout Task Graphs



(a) Baseline (b) Baseline (c) Ray-enabled

Experiment	System	Speedup
(a): Parallel Rollouts on CPU	Ray + TensorFlow	1.0x
(b): Policy Evaluation on GPU	TensorFlow	1.3x
(c): Fine grained rollouts	Ray + TensorFlow	4.1x

- Ray enables the efficient execution of fine-grained task graphs like (c).

Controlling a Simulated Robot in Real-Time

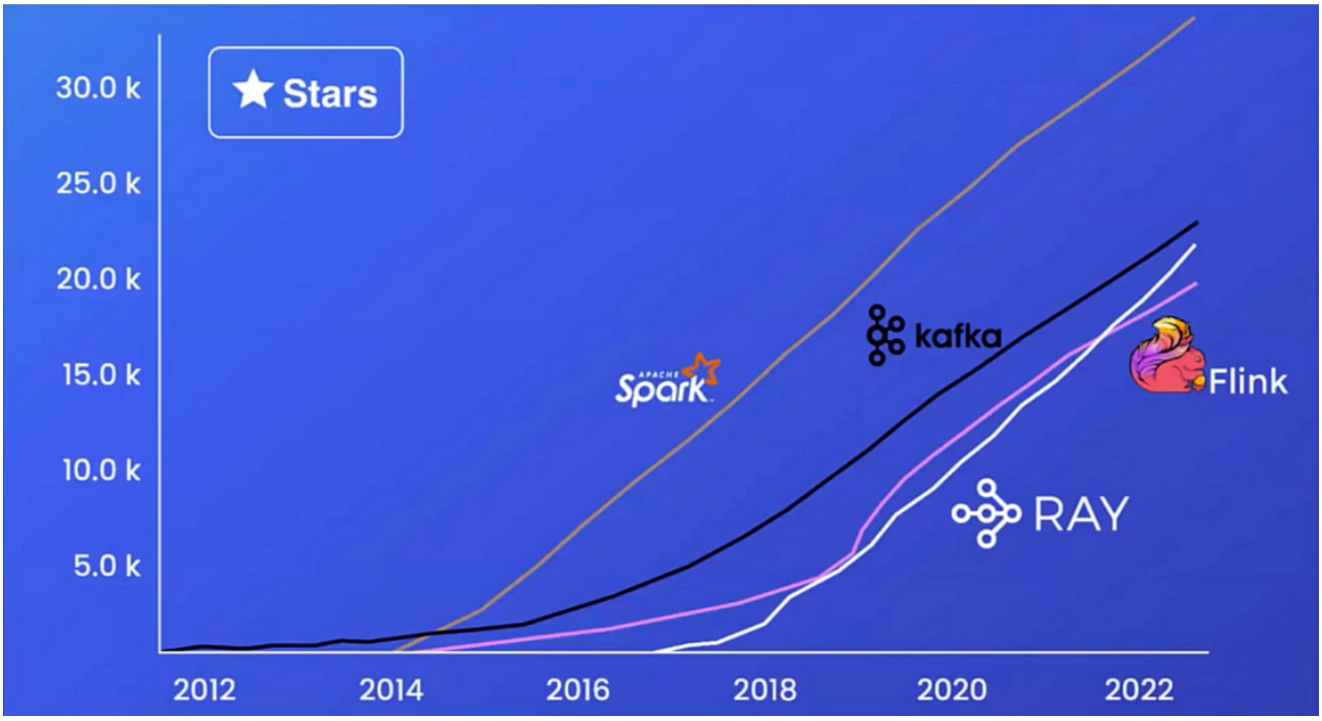
Time budget (ms)	30	20	10	5	3	2	1
Actions fulfilled (%)	100	100	100	100	99.6	60	35
Stable walk?	Yes	Yes	Yes	Yes	Yes	No	No

- Using Ray, we control a simulated robot that has a certain time budget for receiving actions in real time.

Ray's project post at the conference 2017 (Photo courtesy by author)

Above is Ray's project post in 2017. We can see it was elegantly simple but scalably powerful for AI applications.

Ray is a stellar ship, proliferating. It is one of the fastest-growing open sources, as shown by the number of Github stars below.



Ray Github stars growth (Source: January 2023)

Ray is emerging in AI engineering and is an essential tool to scale LLM and RL. Ray is positioned for the massive AI opportunities ahead.