
AI ON
INTEL
intel AI
Ray: Distributed Python for AI
Stephen Offer

What is Ray?

Ray is a fast and simple framework for building and running distributed applications. Specifically targeted for real-time AI applications, Ray is made for low-latency and high-scalability compute.

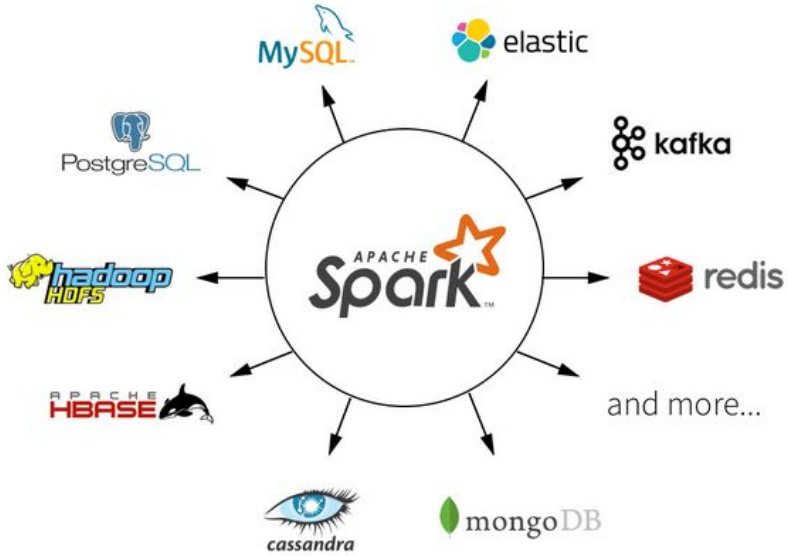
Ray comes from UC Berkeley's RISELab which focuses on Real-time, Intelligent, Secure, and Explainable frameworks.

This framework was created to solve the issues surrounding machine learning software with regards to those acronyms:

- **Real-time**: responds in milliseconds
- **Intelligent**: using sophisticated models/algorithms
- **Secure**: protecting the data and system
- **Explainable**: auditable and verifiable

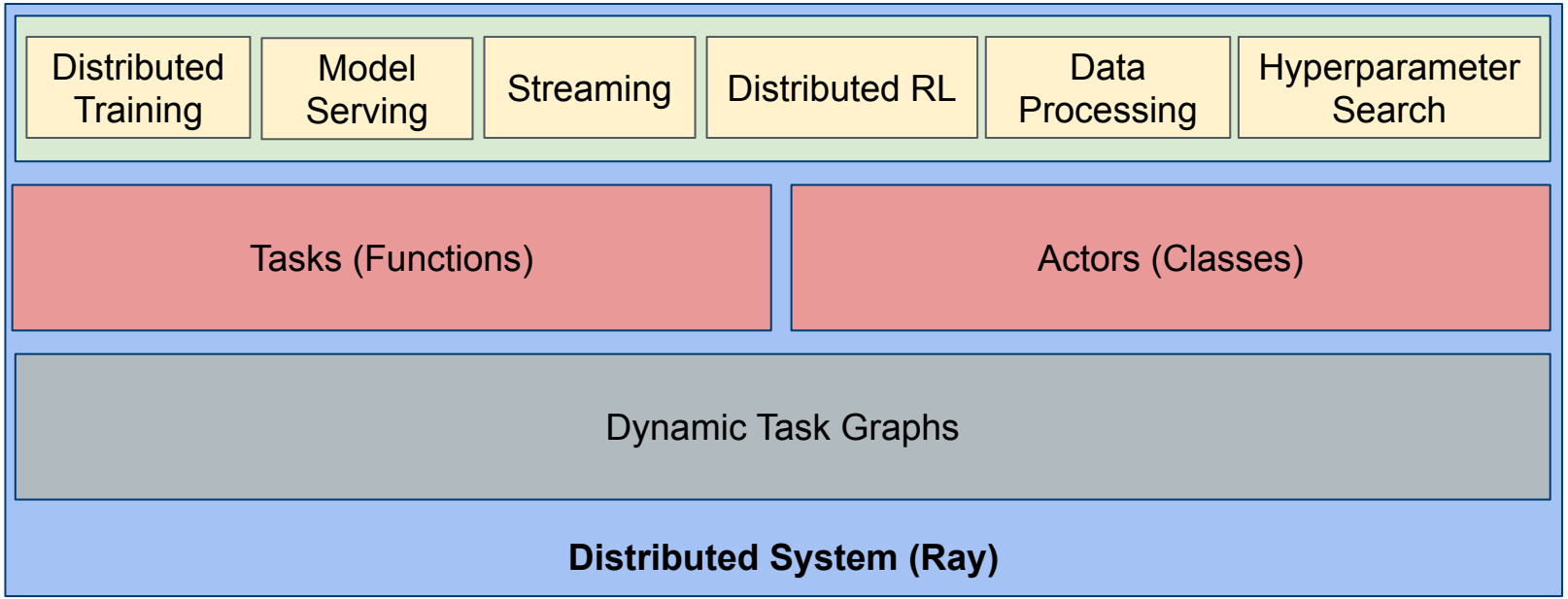
What does Ray solve?

Previous Software Ecosystem

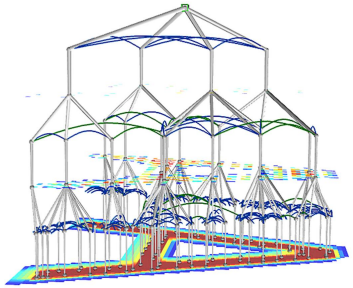


Chainer
mxnet
Caffe2
Microsoft CNTK
TensorFlow
K Keras
GLUON
PYTORCH
theano

What is Ray? A unified machine learning ecosystem.



Supervised Learning



Deep learning poses large compute problems in an ever increasing demand to deploy larger and larger models on vast datasets:

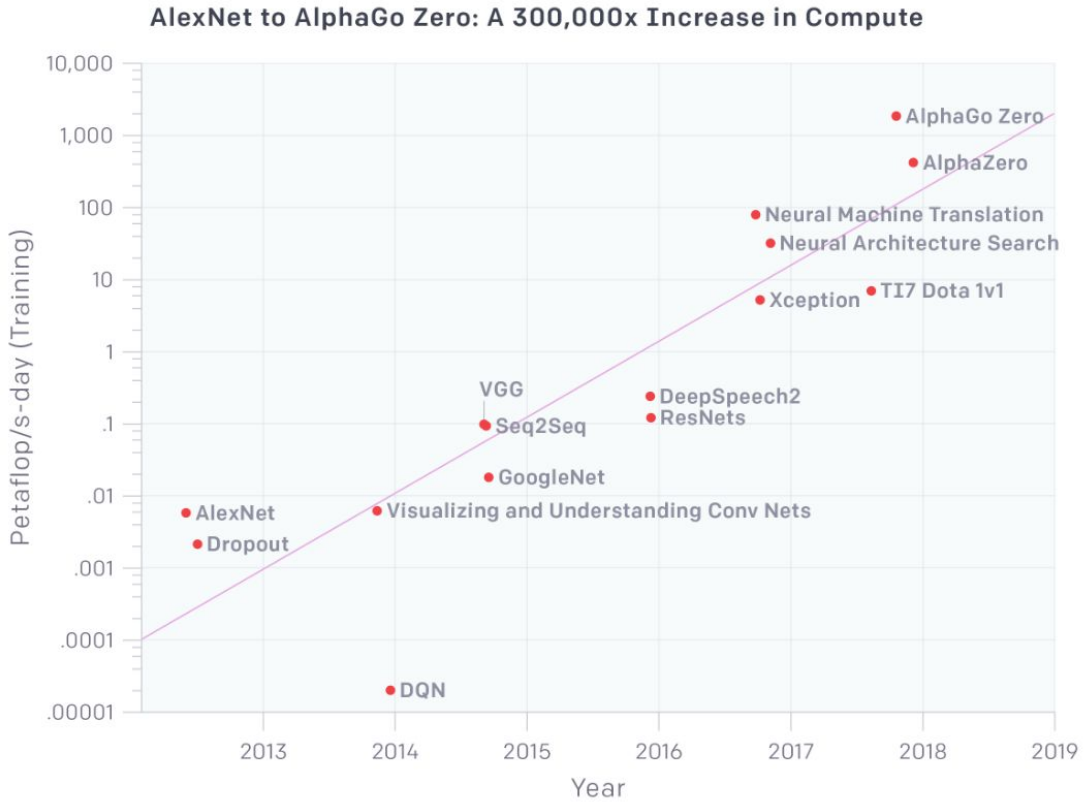
- Traditional training is serial. While there are distributed training algorithms, these are difficult to create and are typically only used in special instances. However, there is a large industrial shift towards distributed deep learning.

- Training can take hours or days or weeks to test out a single configuration of the model, which delays production and causes enormous server costs.

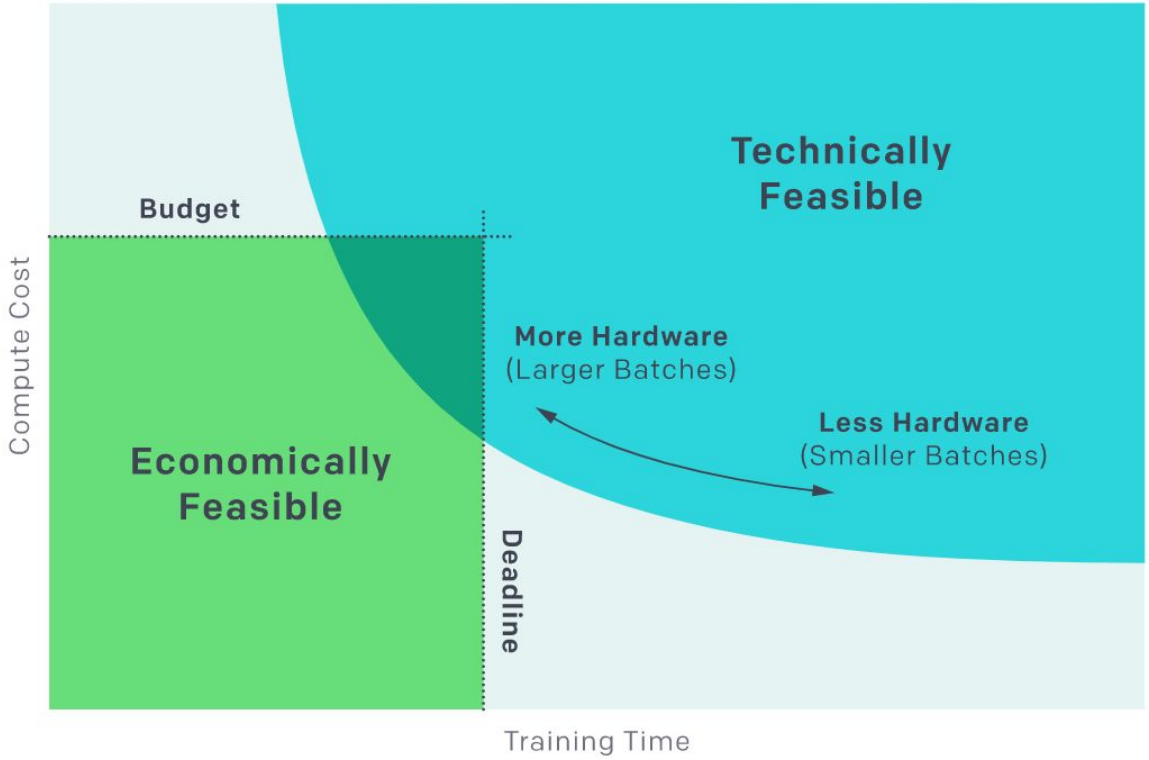
- Getting these types of solutions to scale requires a combination of orchestrators, virtual environments, schedulers, shared memory handlers, etc.. making the transition between serial and distributed deep learning essentially requires an additional team of system and software engineers.

Ray solves these issues by allowing the user to to easily create complex deep learning systems at scale without the additional development team by having all of the components for the actual distribution within a single framework.

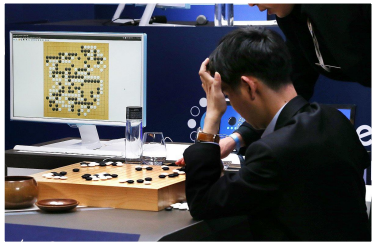
Growing Issues with Machine Learning



Growing Issues with Machine Learning



Reinforcement Learning



Reinforcement learning has vastly different computational requirements than supervised learning. With RL, there are many various requirements for a successful RL framework:

- It must be able to support simulations that might take a few hours or a few milliseconds.
- It must be able to support agent training, which could be anything from a simple PPO agent up to a large ResNet Q-approximation network.
- It must be able to utilize a combination of CPUs and/or accelerators.
- It must be scalable from a few workers to hundreds depending on the algorithm.

Ray is able to fit all of these needs by being a “dynamic computation framework that handles millions of heterogeneous tasks per second at millisecond level latency.”

Framework Comparisons

In comparison to other possible frameworks to use for reinforcement learning or distributed deep learning:

"Bulk-synchronous parallel systems such as MapReduce, Apache Spark, and Dryad do not support simulation or policy serving.

Task-parallel systems such as CIEL and Dask provide little support for distributed training and serving. The same is true for streaming systems such as Naiad and Storm.

Distributed deep-learning frameworks such as TensorFlow and MXNet do not naturally support simulation and serving. Finally, model-serving systems such as TensorFlow Serving and Clipper support neither training nor simulation."

Furthermore, other machine learning frameworks such as PyTorch, PaddlePaddle, and Keras do not currently offer distributed versions. These systems must be made from scratch.

In terms of RL frameworks, there are few RL frameworks that also accomodate distribution.

Distributed Inference

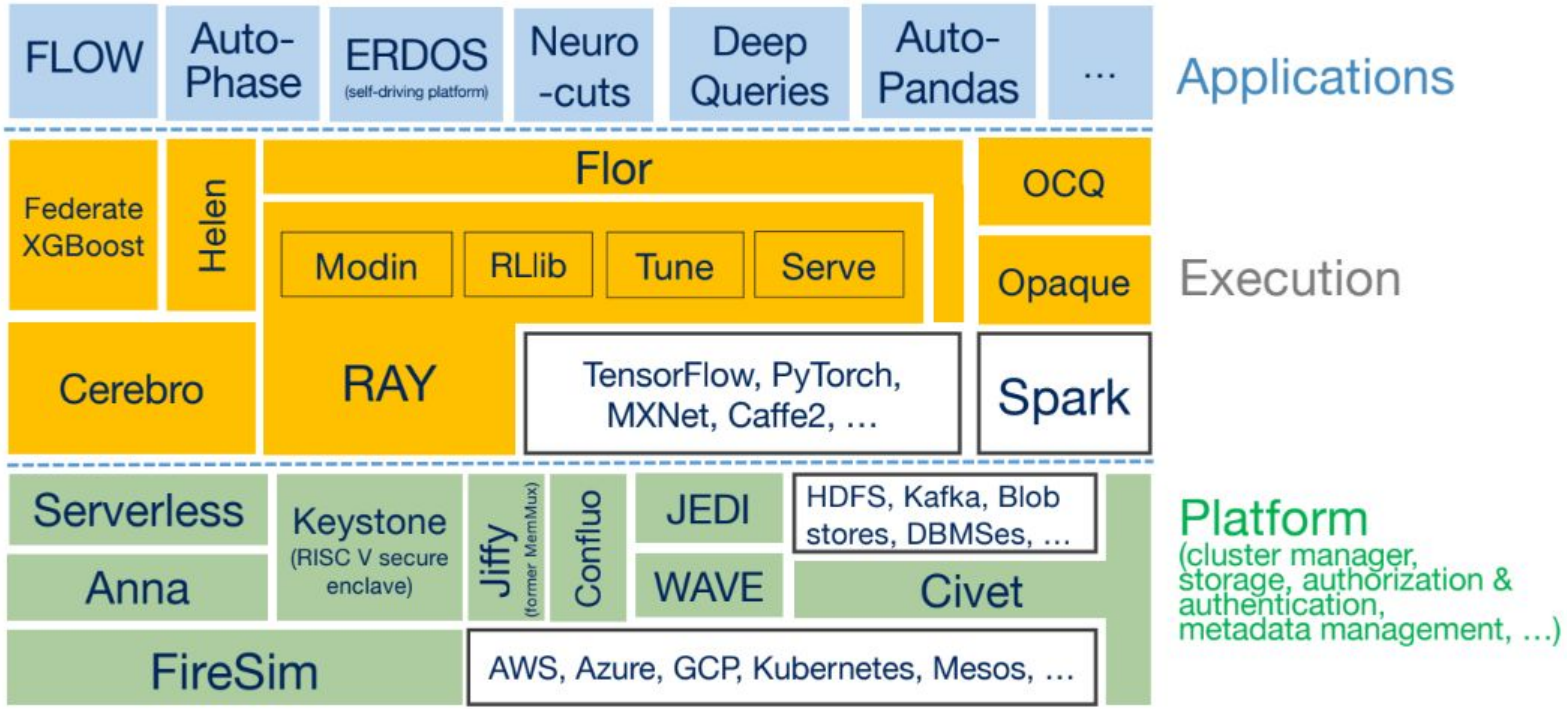
Distributed inference algorithms can be split into two groups, model parallelism and data parallelism.

Model parallelism is where for large models, the layers are partitioned among multiple computational nodes. The activations from the output of a node's model partition are sent to the input of the corresponding node with the next partition of the model. This technique is for large models that cannot fit on one node.

For smaller models, data parallelism can be used. This is where each node has a copy or copies of a model and the data is batched into separate sections and each subset is sent to another copy of the model. The output of each copy is then sent to a parameter server and pieced together so the results are in the correct order.

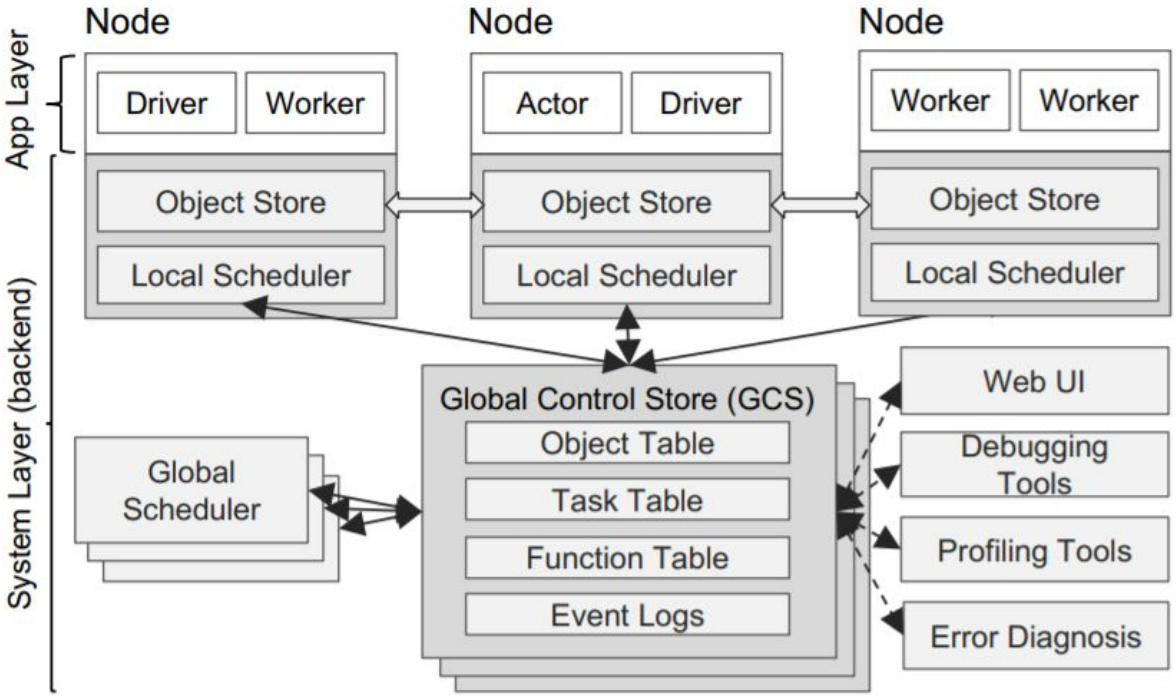
Ray's ability to create stateful actors make it very simple to create these types of systems and have been seen to greatly improve model performance.

RISE Stack

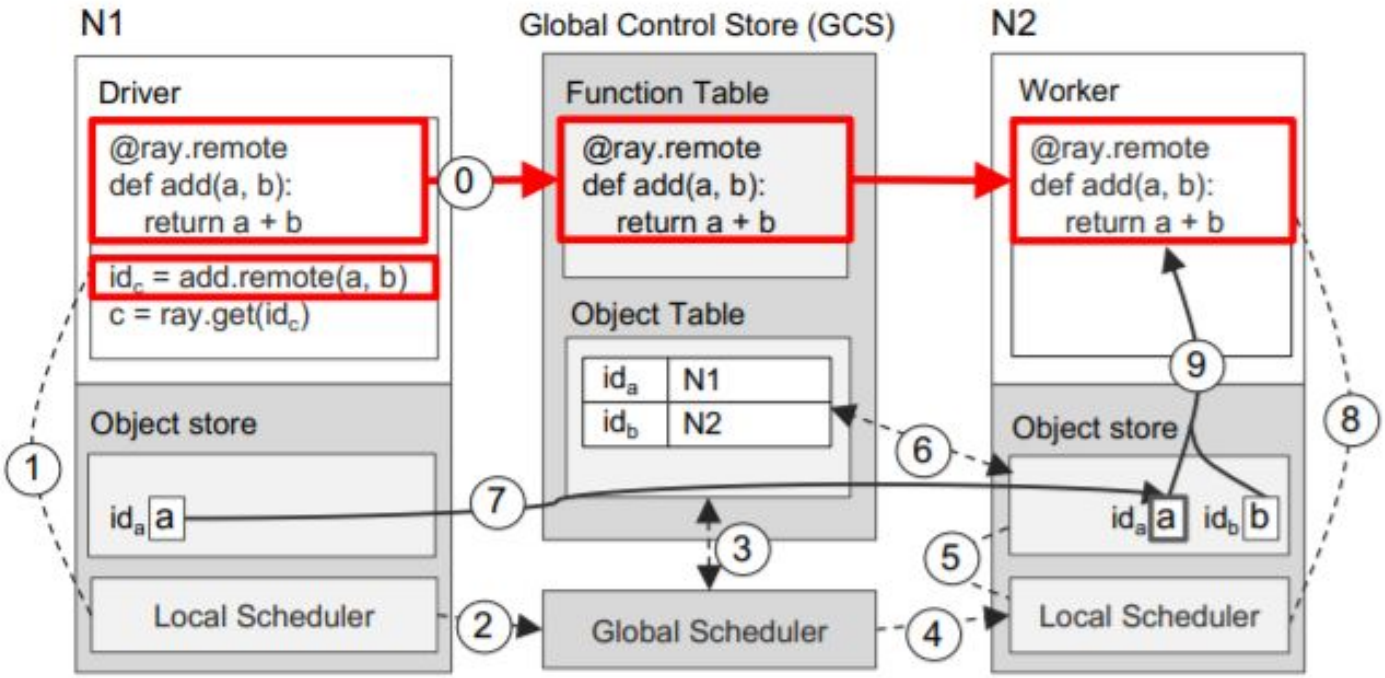

FLOW
Auto-Phase
ERDOS
(self-driving platform)
Neuro-cuts
Deep Queries
Auto-Pandas
…
Applications
Federate XGBoost
Helen
Flor
Modin
RLlib
Tune
Serve
OCQ
Opaque
Execution
Cerebro
RAY
TensorFlow, PyTorch, MXNet, Caffe2, …
Spark
Serverless
Anna
Keystone
(RISC V secure enclave)
Jiffy
(former MemMux)
Confluo
JEDI
WAVE
HDFS, Kafka, Blob stores, DBMSes, …
Civet
FireSim
AWS, Azure, GCP, Kubernetes, Mesos, …
Platform
(cluster manager, storage, authorization & authentication, metadata management, …)

How does Ray work?

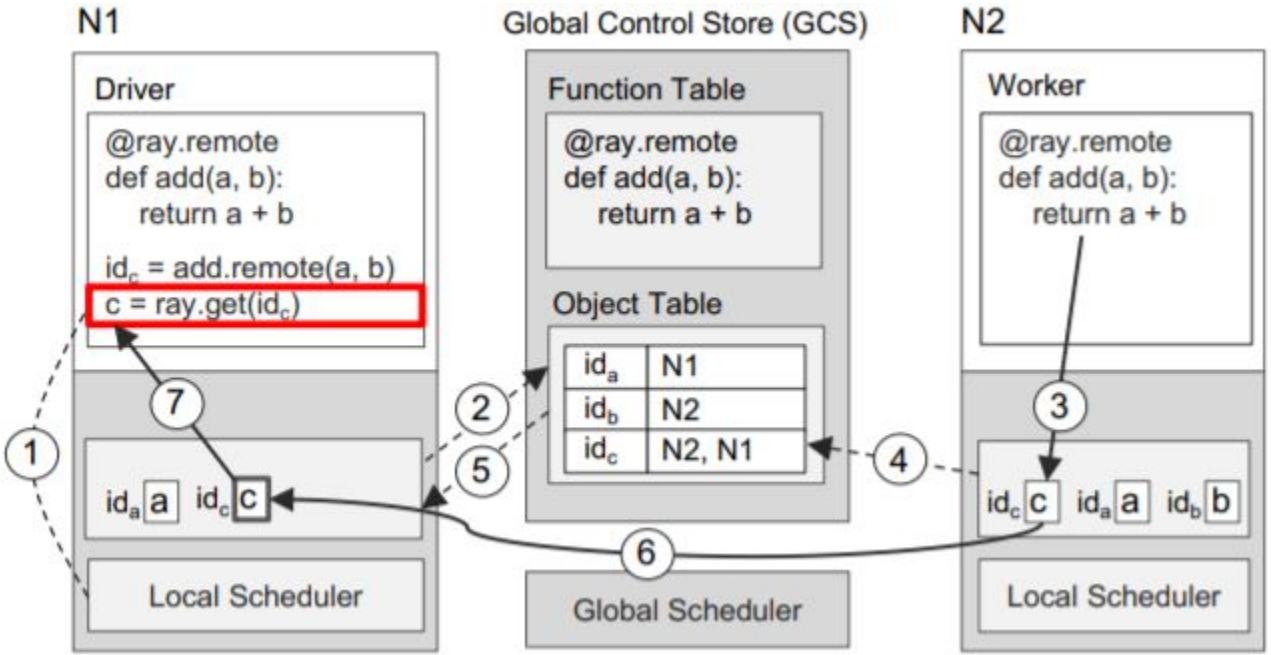
System Architecture



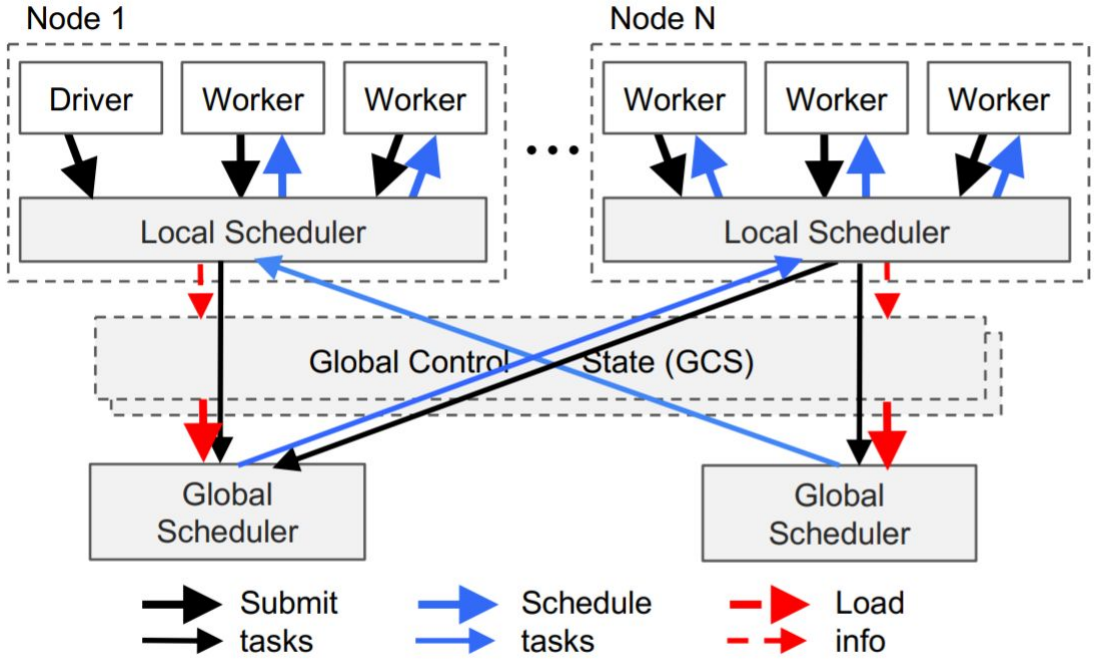
Task Example Part 1



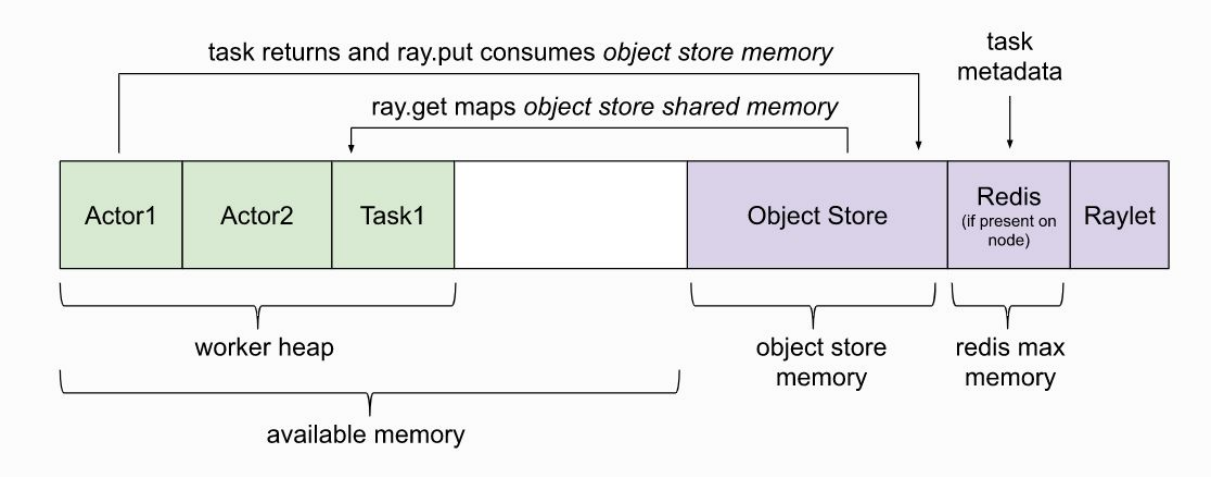
Task Example Part 2



Bottom-up Distributed Scheduler



Memory



Ray system memory: this is memory used internally by Ray

- **Redis**: memory used for storing task lineage and object metadata.
- **Raylet**: memory used by the C++ raylet process running on each node.

Application memory: this is memory used by your application

- **Worker heap**: memory used by your application, essentially the resident set size minus the shared memory usage.
- **Object store memory**: memory used when your applications creates objects in the objects store via ray.put and when returning values from remote functions.
- **Object store shared memory**: memory used when your applications reads objects via ray.get.

Tasks

Ray enables arbitrary Python functions to be executed asynchronously. These asynchronous Ray functions are called "remote functions". The standard way to turn a Python function into a remote function is to add the @ray.remote decorator. Here is an example.

```
# A regular Python function.
def regular function():
    return 1

# A Ray remote function.
@ray.remote
def remote_function():
     return 1
```

Actors

```
@ray.remote
class Foo(object):

    @ray.method(num_return_vals=2)
    def bar(self):
        return 1, 2

f = Foo.remote()

obj_id1, obj_id2 = f.bar.remote()
assert ray.get(obj_id1) == 1
assert ray.get(obj_id2) == 2
```

An actor represents a stateful computation. Each actor exposes methods that can be invoked remotely and are executed serially.

A method execution is similar to a task, in that it executes remotely and returns a future, but differs in that it executes on a stateful worker.

What are the advantages of Ray?

Ray has the advantage over other frameworks in that:

- It is the first distributed framework that unifies training, simulation, and serving— necessary components of emerging applications.
- Since the control state is stored in a sharded metadata store and all other components are stateless, it is highly scalable and fault tolerant.
- The bottom-up distributed scheduling strategy also makes it highly scalable.
- Developers can test on small devices like PCs or laptops and scale up to a large data center with changing only one line of code.
- Unlike many distributed frameworks, it does not require virtual environments, separate orchestrators, etc..
- Simple pip installation with minimal dependencies.

What is Ray not?

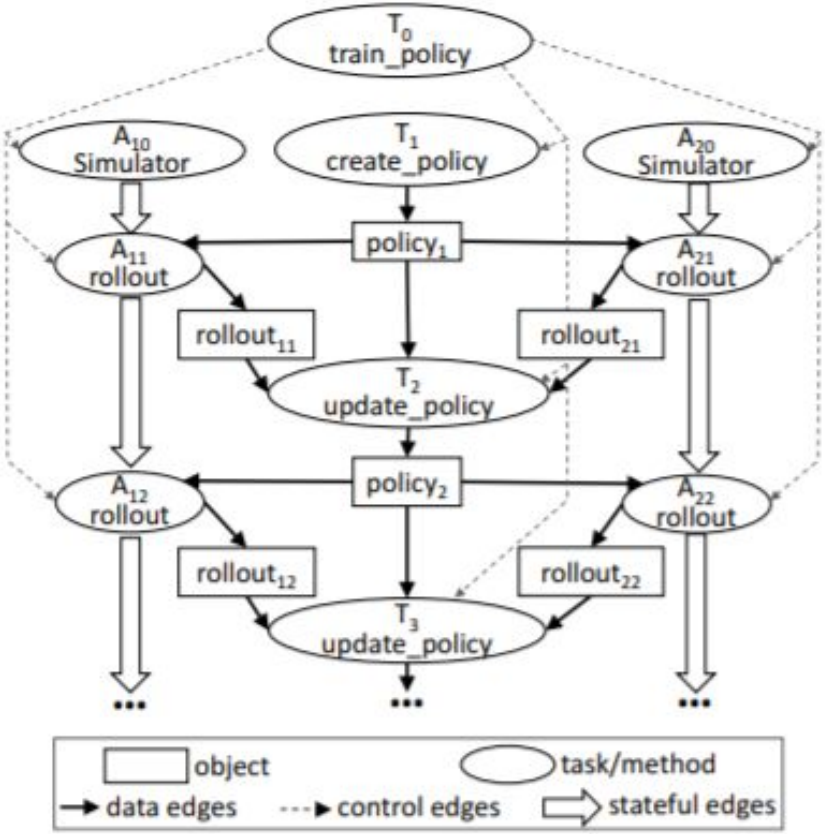
While Ray can be used for many types of applications, it is important to clarify when to use other frameworks that may border the same functionality of Ray:

- It has a centralized architecture and not decentralized, so it cannot adapt to the head node closing.
- It currently does not support Windows.
- It only communicates through one level of a network tree (not IoT distributed communication).
- It is still a growing framework and does have as much functionality as more mature frameworks (i.e. does not address straggler mitigation or query optimization).
- It does not yet have as large of a community around the framework due to it being relatively new.

RL Example

```
// evaluate policy by interacting with env. (e.g., simulator)
rollout(policy, environment):
    trajectory = []
    state = environment.initial_state()
    while (not environment.has_terminated()):
        action = policy.compute(state) // Serving
        state, reward = environment.step(action) // Simulation
        trajectory.append(state, reward)
    return trajectory

// improve policy iteratively until it converges
train_policy(environment):
    policy = initial_policy()
    while (policy has not converged):
        trajectories = []
        for i from 1 to k:
            // evaluate policy by generating k rollouts
            trajectories.append(rollout(policy, environment))
            // improve policy
        policy = policy.update(trajectories) // Training
    return policy
```



Build with Ray

RLlib

RLlib has extra dependencies on top of `ray`. First, you'll need to install either PyTorch or TensorFlow. Then, install the RLlib module:

```
pip install ray[rllib]   # also recommended: ray[debug]
```

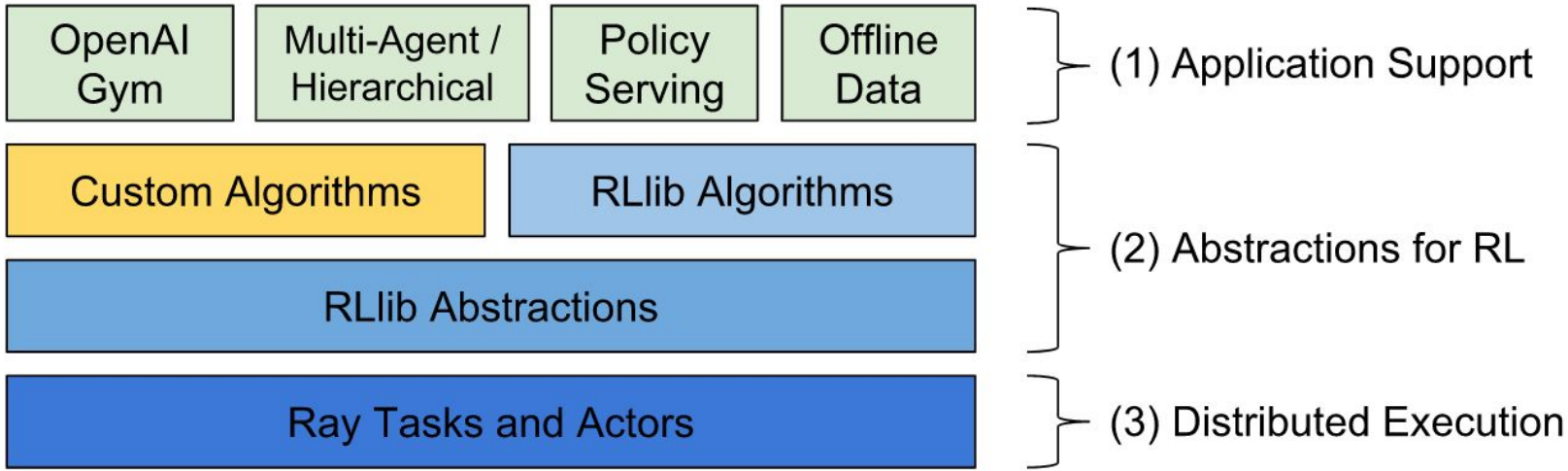
Then, you can try out training in the following equivalent ways:

```
rllib train --run=PPO --env=CartPole-v0
```

```
from ray import tune
from ray.rllib.agents.ppo import PPOTrainer
tune.run(PPOTrainer, config={"env": "CartPole-v0"})
```

RLlib

RLlib is an open-source library for reinforcement learning that offers both high scalability and a unified API for a variety of applications. RLlib natively supports TensorFlow, TensorFlow Eager, and PyTorch, but most of its internals are framework agnostic.



Tune

Tune is a Python library for hyperparameter tuning at any scale. Core features:

- Launch a multi-node distributed hyperparameter sweep in less than 10 lines of code.
- Supports any machine learning framework, including PyTorch, XGBoost, MXNet, and Keras.
- Visualize results with TensorBoard.
- Choose among scalable SOTA algorithms such as Population Based Training (PBT), Vizier's Median Stopping Rule, HyperBand/ASHA.
- Tune integrates with many optimization libraries such as Facebook Ax, HyperOpt, and Bayesian Optimization and enables you to scale them transparently.

Tune


```
import torch.optim as optim
from ray import tune
from ray.tune.examples.mnist_pytorch import get_data_loaders, ConvNet, train, test

def train_mnist(config):
    train_loader, test_loader = get_data_loaders()
    model = ConvNet()
    optimizer = optim.SGD(model.parameters(), lr=config["lr"])
    for i in range(10):
        train(model, optimizer, train_loader)
        acc = test(model, test_loader)
        tune.track.log(mean_accuracy=acc)

analysis = tune.run(train_mnist, config={"lr": tune.grid_search([0.001, 0.01, 0.1])})

print("Best config: ", analysis.get_best_config(metric="mean_accuracy"))

# Get a dataframe for analyzing trial results.
df = analysis.dataframe()
```

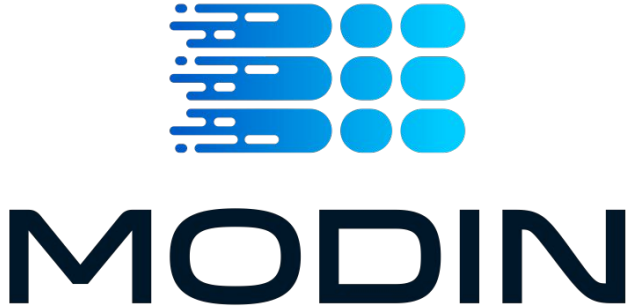
Modin

Modin uses Ray to provide an effortless way to speed up your pandas notebooks, scripts, and libraries. Unlike other distributed DataFrame libraries, Modin provides seamless integration and compatibility with existing pandas code. Even using the DataFrame constructor is identical.

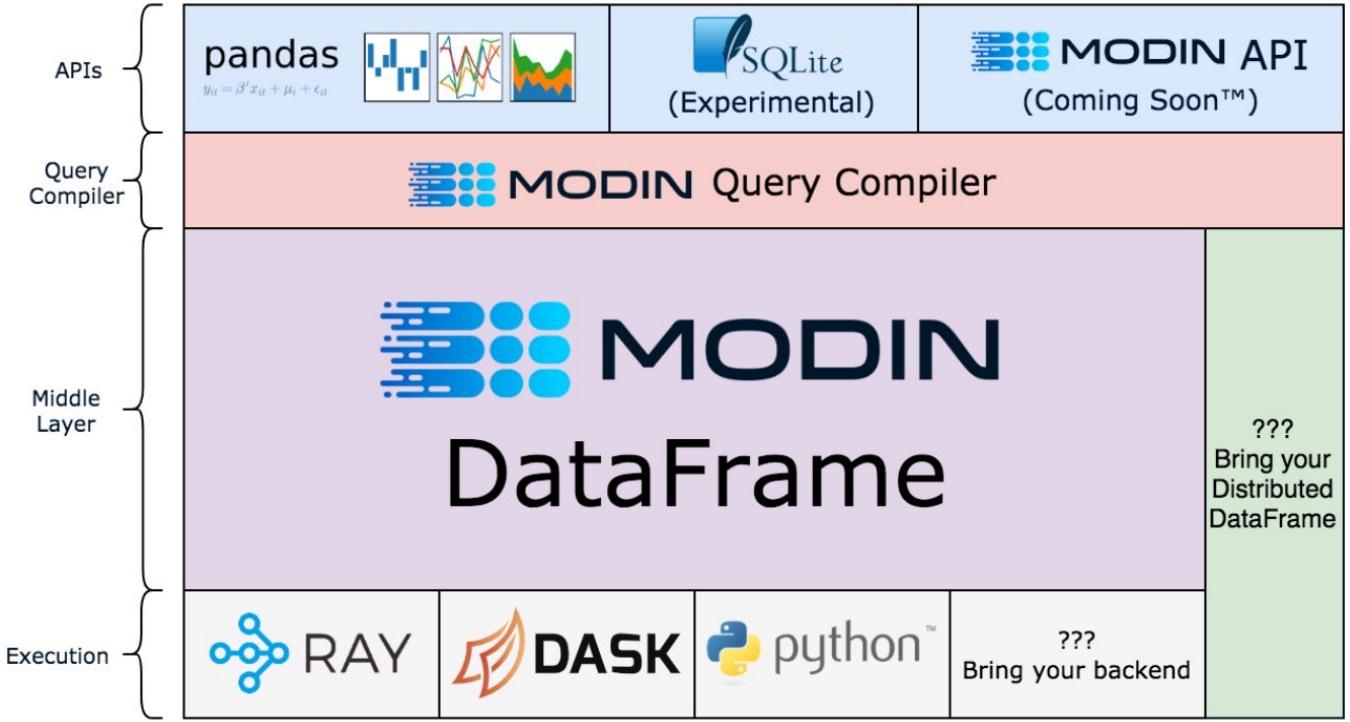
```
>>> pip install modin
```

```
import modin.pandas as pd
import numpy as np

frame data = np.random.randint(0, 100,
    size=(2**10, 2**8))
df = pd.DataFrame(frame_data)
```



Modin


APIs
pandas
SQLite
(Experimental)
MODIN API
(Coming Soon™)
Query Compiler
MODIN Query Compiler
Middle Layer
MODIN
DataFrame
???
Bring your Distributed DataFrame
Execution
RAY
DASK
python
???
Bring your backend

Framework Building Example: dKeras

dKeras is a distributed Keras engine that is built on top of Ray. By wrapping dKeras around your original Keras model, it allows you to use many distributed deep learning techniques to automatically improve your system's performance.

With an easy-to-use API and a backend framework that can be deployed from the laptop to the data center, dKeras simplifies what used to be a complex and time-consuming process into only a few adjustments.

```
from tensorflow.keras.applications import ResNet50
from dkeras import dKeras
import numpy as np
import ray

ray.init()

data = np.random.uniform(-1, 1, (100, 224, 224, 3))

model = dKeras(ResNet50, init_ray=False, wait_for_workers=True, n_workers=4)
preds = model.predict(data)
```

Conclusion

Ray is a fast and simple framework for building and running distributed applications.

It solves many of the technical issues with creating modern systems by providing a unified platform for:

- Data analytics
- Distributed processing
- Machine learning/deep learning
- Reinforcement learning
- Hyperparameter search
- Streaming
- Model serving

While a new framework, it is fast growing and already has many frameworks built on top of it such as:

- RLlib
- Modin
- Tune