Ray API: Tasks & Actors

DS 5110: Big Data Systems

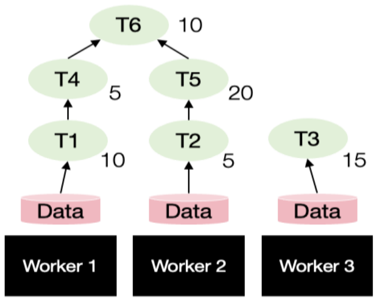
Spring 2025

Lecture 9

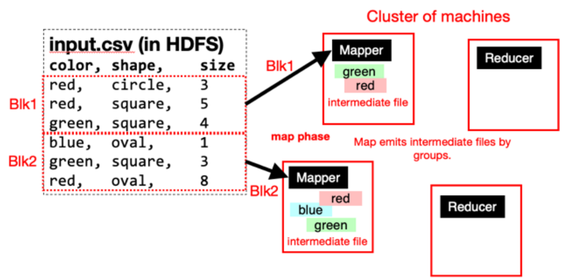
Yue Cheng

University of Virginia

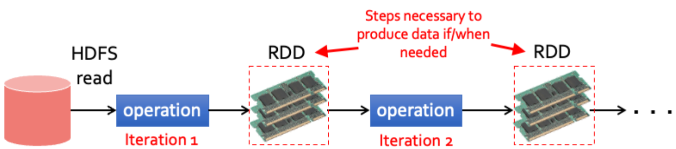
A recap of big data systems covered so far…



Dask: Exposes APIs that automatically parallelize Python analytics programs to a cluster of workers

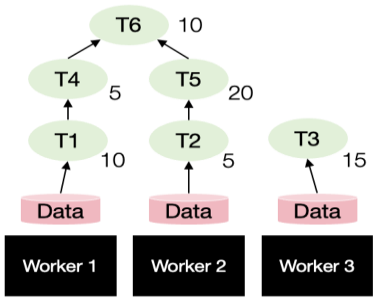


MapReduce: Developers program Map and Reduce to implement batch processing applications

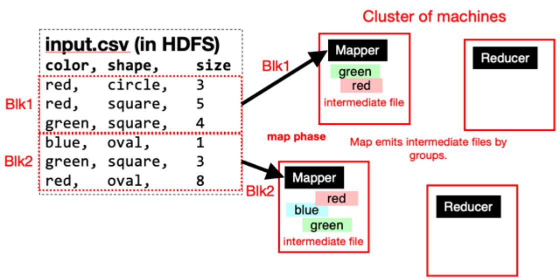


Spark: Based on MapReduce, but with extensive perf optimizations and a much richer set of programming APIs

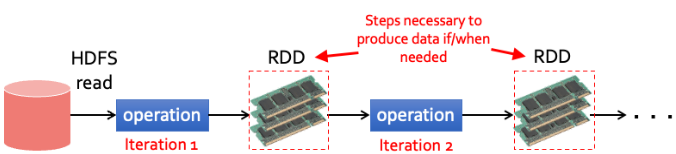
A recap of big data systems covered so far...



Dask: Exposes APIs that automatically parallelize Python analytics programs to a cluster of workers



MapReduce: Developers program Map and Reduce to implement batch processing applications



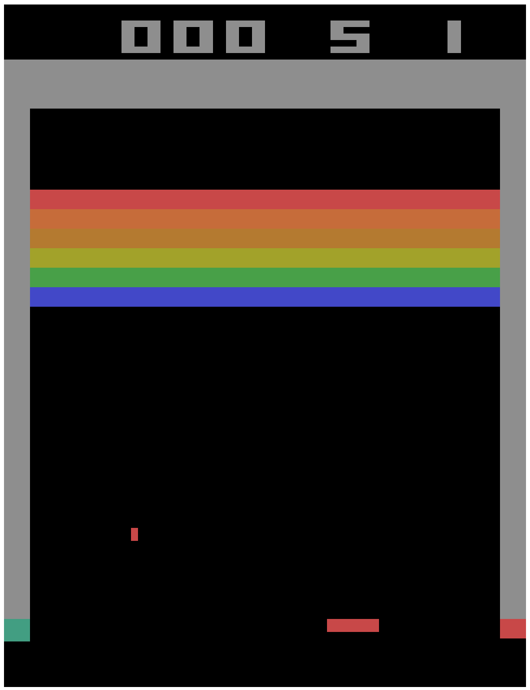
Spark: Based on MapReduce, but with extensive perf optimizations and a much richer set of programming APIs

Ray is different from all the others that we covered...

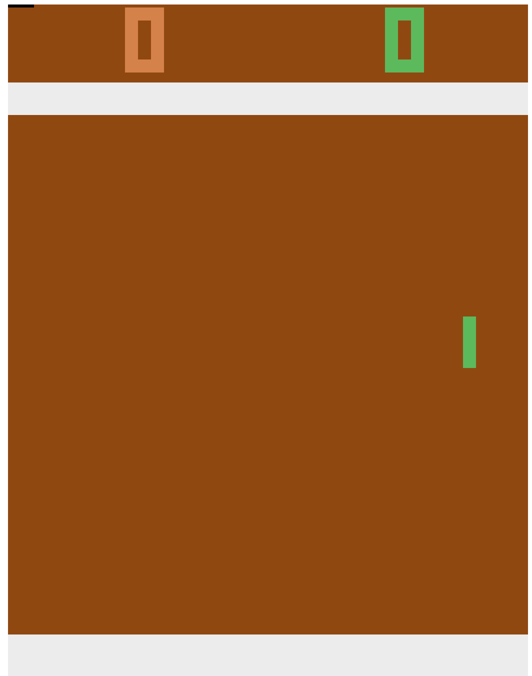
Learning objectives

- Know the unique requirements of RL applications and the motivation behind Ray
- Understand the difference of two Ray APIs: tasks and actors

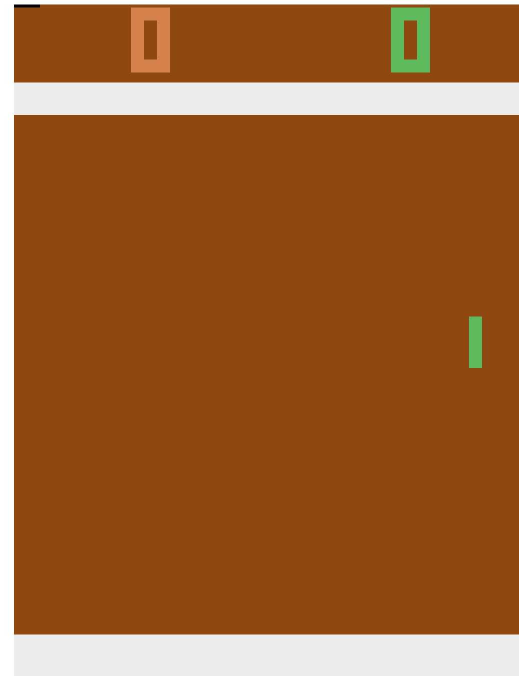
Motivation: Reinforcement learning



Atari breakout



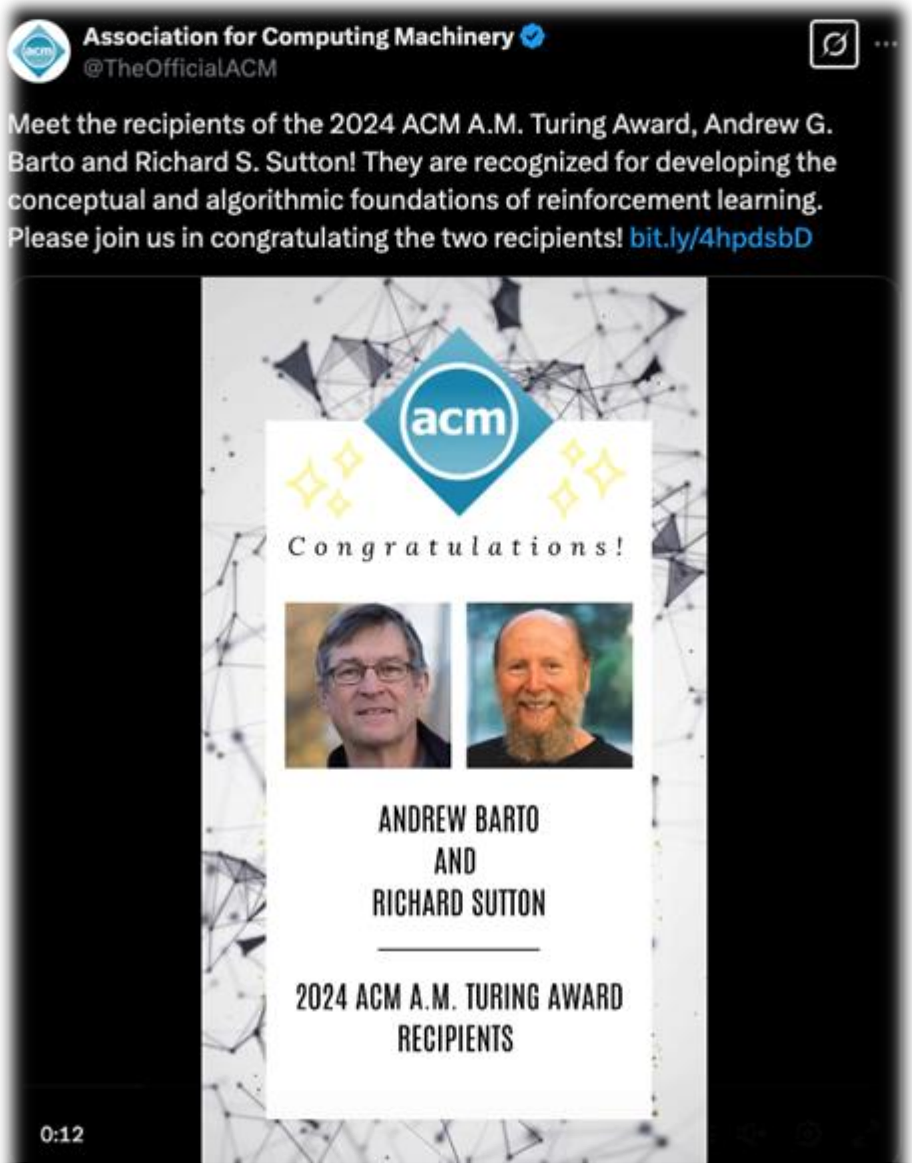
Pong: after 30 mins of training



Pong: DQN wins like a boss

*: Playing Atari with Deep Reinforcement Learning: https://arxiv.org/abs/1312.5602

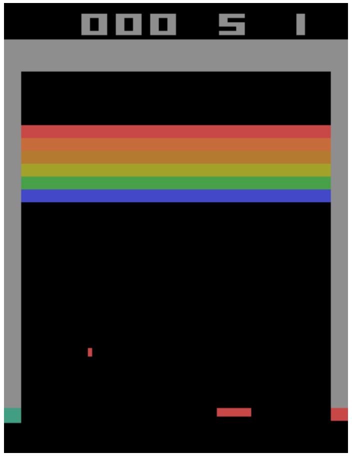
Motivation: Reinforcement learning

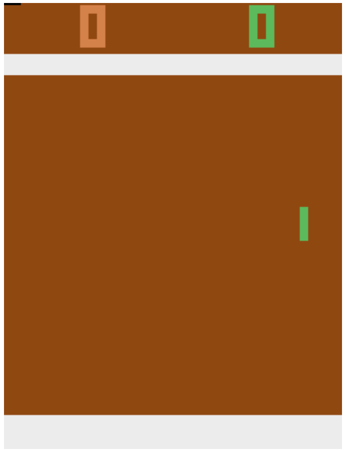


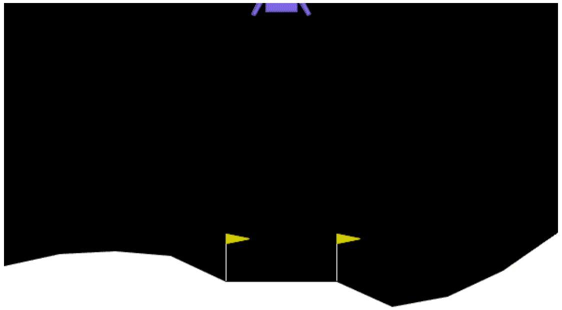
RL application pattern

- Process inputs from different sensors (sources) in parallel & real-time

- Execute large number of simulations, e.g., up to 100s of millions







RL setup

Agent

Policy:

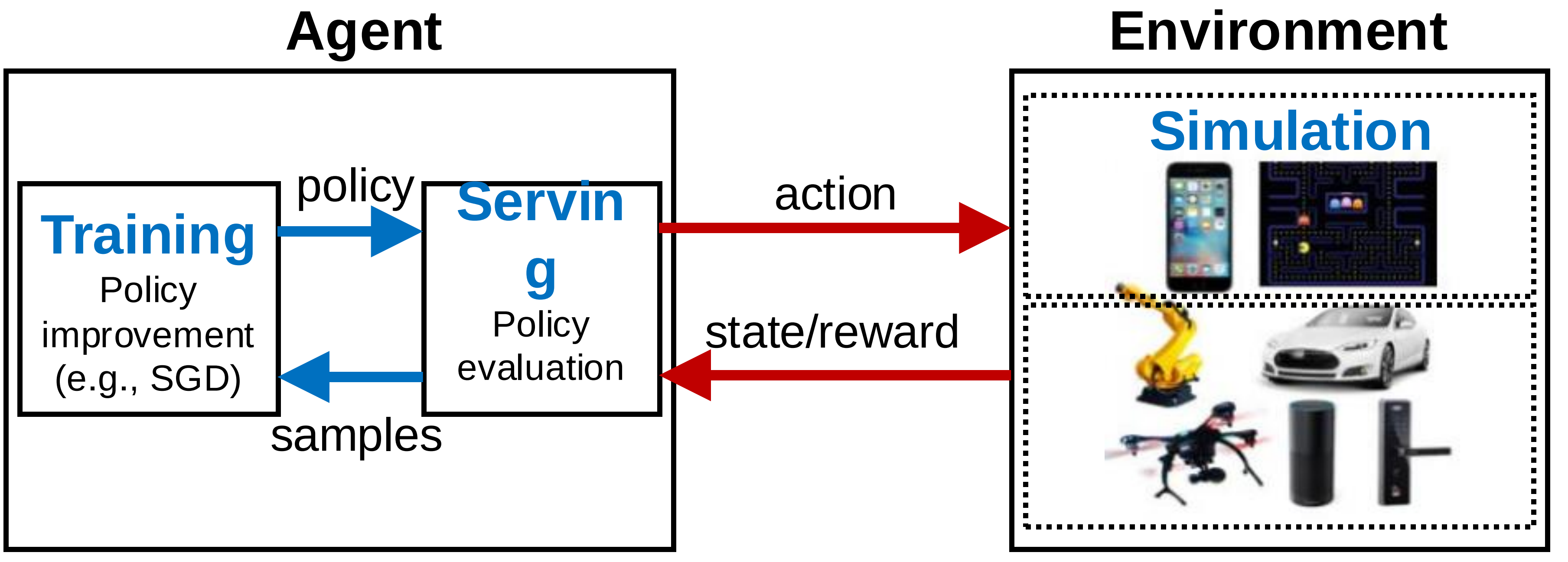
state → action

action

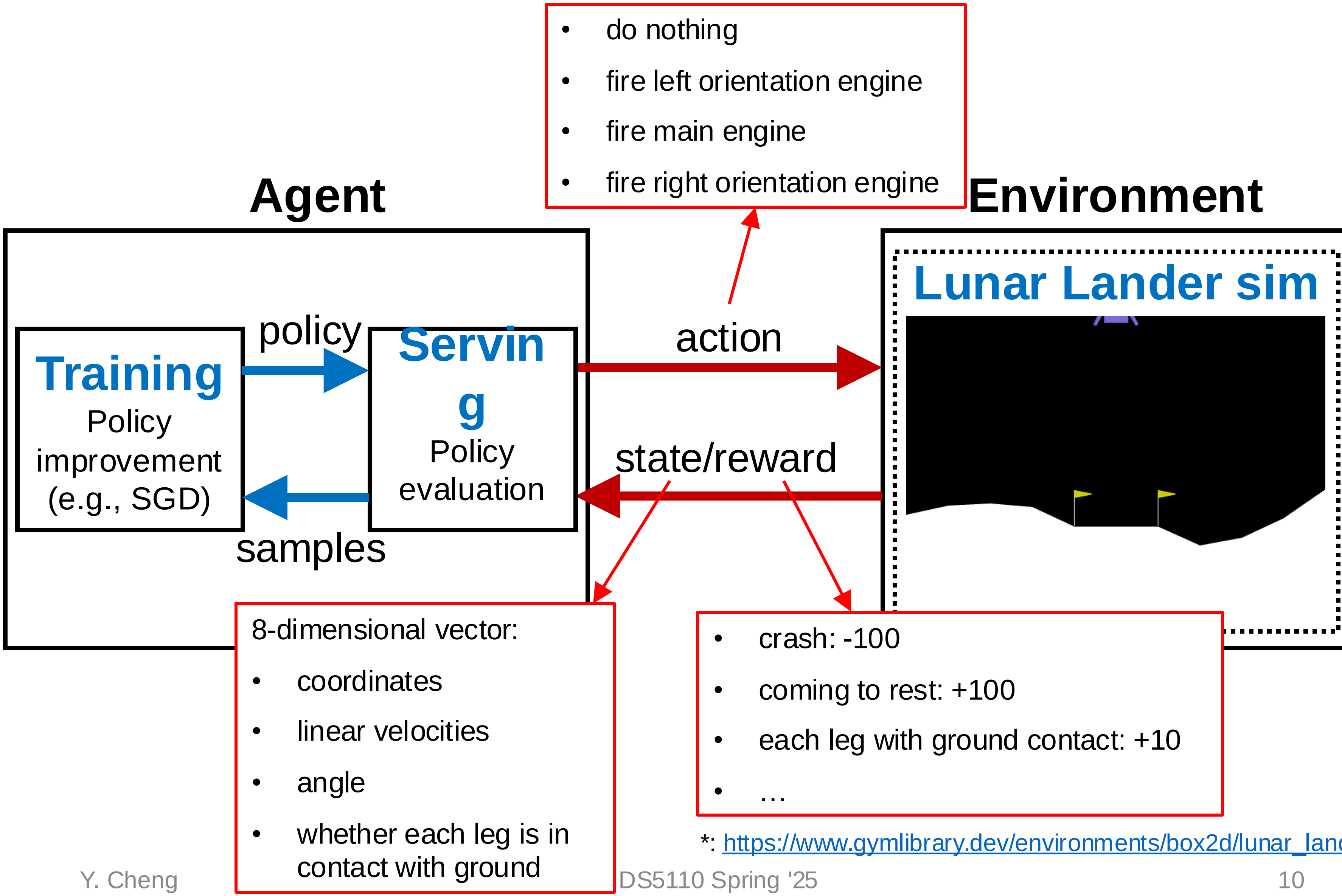
state/reward

Environment

RL setup zoomed in

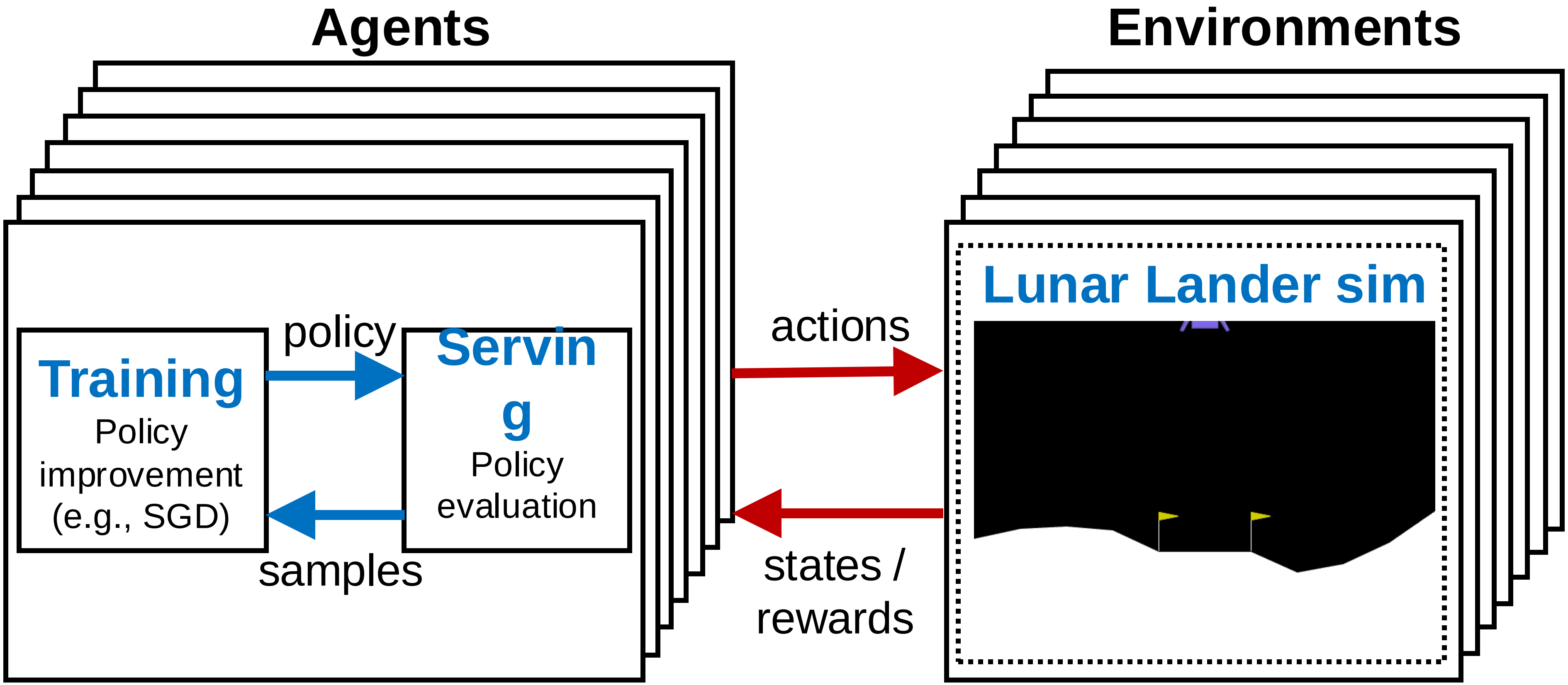


RL setup zoomed in (Lunar Lander)



*: https://www.gymlibrary.dev/environments/box2d/lunar_lander/

Scaling out the RL setup



RL application pattern

- Process inputs from different sensors (sources) in parallel & real-time
- Execute large number of simulations, e.g., up to 100s of millions
- Simulation outcomes are used to update policy (e.g., Q-learning/SGD/Adam)

RL application requirements

- Need to handle dynamic task graphs, where tasks have:
  - heterogeneous durations (seconds to minutes)
  - heterogeneous computations (CPUs vs. GPUs)

- Need to schedule millions of tasks / sec

- Need to make it easy to parallelize ML algorithms (in Python)

Today's AI/ML data system landscape

Distributed systems	Distributed systems	Distributed systems	Distributed systems
Data processing	Model training	Model tuning	Model serving
Spark, Hadoop, Dask, Modin, …	PyTorch, TensorFlow, scikit-learn, …	Optuna, Hyperopt, SigOpt, MLflow, …	vLLM, FastAPI, Arize, Alibi, …

Today's AI/ML data system landscape

Distributed systems	Distributed systems	Distributed systems	Distributed systems
Data processing	Model training	Model tuning	Model serving
Spark, Hadoop, Dask, Modin, …	PyTorch, TensorFlow, scikit-learn, …	Optuna, Hyperopt, SigOpt, MLflow, …	vLLM, FastAPI, Arize, Alibi, …

Emerging AI applications require **stitching** together **multiple** disparate systems

Ad hoc integrations are difficult to manage and program!

Ray ecosystem offers a unified solution

Distributed systems	Distributed systems	Distributed systems	Distributed systems
Data processing	Model training	Model tuning	Model serving
Spark, Hadoop, Dask, Modin, …	PyTorch, TensorFlow, scikit-learn, …	Optuna, Hyperopt, SigOpt, MLflow, …	vLLM, FastAPI, Arize, Alibi, …

Ray AI Runtime (AIR)

Data processing Ray Dataset	**Model training** Ray Training Ray RLlib	**Model tuning** Ray Tune	**Model serving** Ray Serve

Ray Core
(remote tasks, actors, scheduling, data sharing, etc.)

Example: Retrieving a data item

```
database = [
  “learning”,
  “Ray”,
  “for”,
  “distributed”,
  “data”,
  “processing”
]
```

```
def retrieve(item_idx):
    time.sleep(item_idx / 10.)
    return item_idx, database[item_idx]
```

```
data = [retrieve(idx) for idx in range(6)]
```

Example: Retrieving a data item

```
database = [
  “learning”,
  “Ray”,
  “for”,
  “distributed”,
  “data”,
  “processing”
]
```

```
def retrieve(item_idx):
    time.sleep(item_idx / 10.)
    return item_idx, database[item_idx]
```

```
0
data = [retrieve(idx) for idx in range(6)]
```

Example: Retrieving a data item

```
database = [
  “learning”,
  “Ray”,
  “for”,
  “distributed”,
  “data”,
  “processing”
]
```

```
def retrieve(item_idx):
    time.sleep(item_idx / 10.)
    return item_idx, database[item_idx]
```

0

```
data = [retrieve(idx) for idx in range(6)]
```

0, “learning”

Example: Retrieving a data item

```
database = [
  “learning”,
  “Ray”,
  “for”,
  “distributed”,
  “data”,
  “processing”
]
```

```
def retrieve(item_idx):
    time.sleep(item_idx / 10.)
    return item_idx, database[item_idx]
```

1

```
data = [retrieve(idx) for idx in range(6)]
```

Example: Retrieving a data item

```
database = [
  “learning”,
  “Ray”,
  “for”,
  “distributed”,
  “data”,
  “processing”
]
```

```
def retrieve(item_idx):
    time.sleep(item_idx / 10.)
    return item_idx, database[item_idx]
```

1

```
data = [retrieve(idx) for idx in range(6)]
```

Example: Retrieving a data item

```
database = [
  “learning”,
  “Ray”,
  “for”,
  “distributed”,
  “data”,
  “processing”
]
```

```
def retrieve(item_idx):
    time.sleep(item_idx / 10.)
    return item_idx, database[item_idx]
```

1

```
data = [retrieve(idx) for idx in range(6)]
```

1, “Ray”

Example: Retrieving a data item

```
database = [
  “learning”,
  “Ray”,
  “for”,
  “distributed”,
  “data”,
  “processing”
]
```

```
def retrieve(item_idx):
    time.sleep(item_idx / 10.)
    return item_idx, database[item_idx]
```

```
5
data = [retrieve(idx) for idx in range(6)]
```

```
5, “processing”
```

Expect a runtime of around (0+1+2+3+4+5)/10 = 1.5 seconds

Ray API: Remote Ray tasks

```
database = [
  “learning”,
  “Ray”,
  “for”,
  “distributed”,
  “data”,
  “processing”
]
```

Ray task decorator

```
def retrieve(item_idx):
    time.sleep(item_idx / 10.)
    return item_idx, database[item_idx]

@ray.remote
def retrieve_task(item_idx):
    return retrieve(item_idx)
```

```
obj_refs = [
      retrieve_task.remote(idx) for idx in range(6)
]
data = ray.get(obj_refs)
```

Ray tasks are **decorated Python functions** that can execute **remotely**.

`task.remote()` executes a task remotely asynchronously and immediately returns a `future` (i.e., an object reference, which you need to explicitly ask the result of).

`ray.get(ObjRef)` fetches the computed result of a remote task referenced by `ObjRef`.

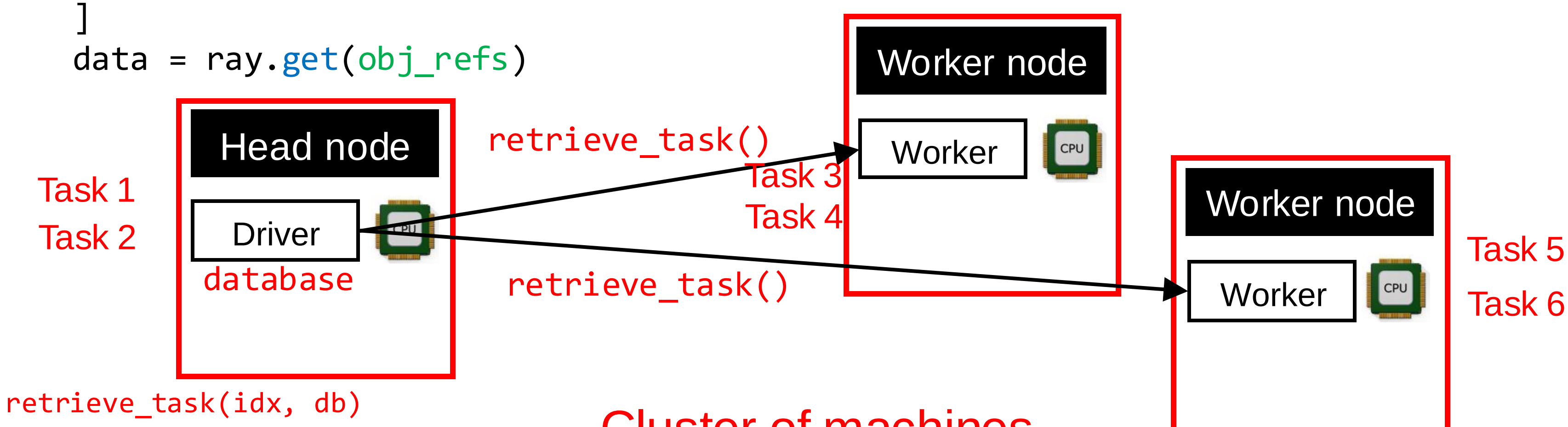
Ray API: Remote Ray tasks

```
database = [
  “learning”,
  “Ray”,
  “for”,
  “distributed”,
  “data”,
  “processing”
]
```

```
def retrieve(item_idx):
    time.sleep(item_idx / 10.)
    return item_idx, database[item_idx]

@ray.remote
def retrieve_task(item_idx):
    return retrieve(item_idx)
```

```
obj_refs = [
      retrieve_task.remote(idx) for idx in range(6)
]
data = ray.get(obj_refs)
```



Cluster of machines

Ray API: Remote Ray tasks

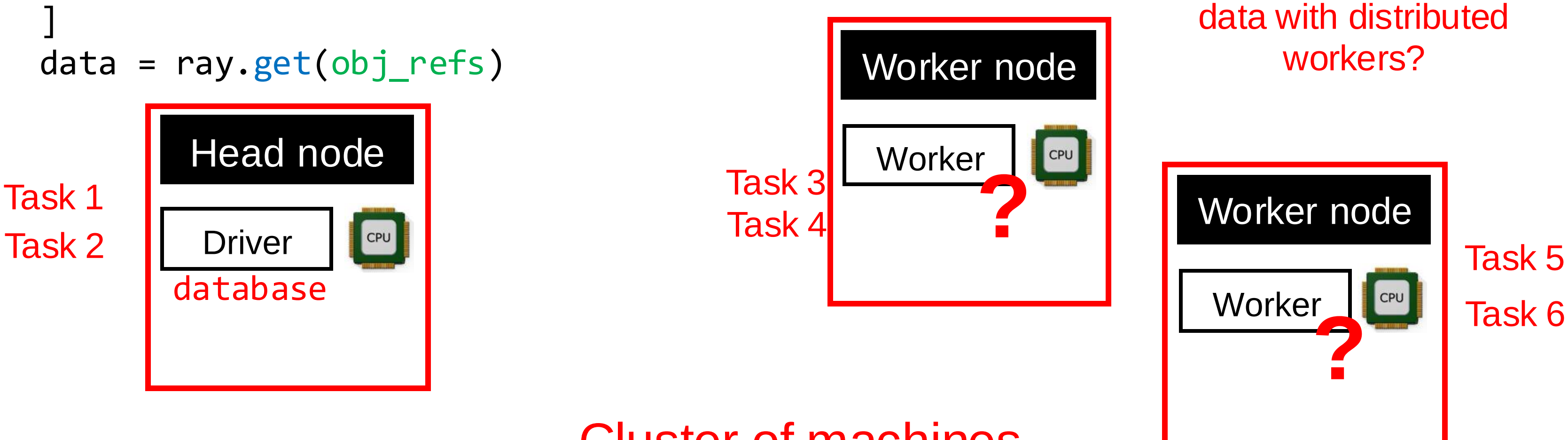
```
database = [
  “learning”,
  “Ray”,
  “for”,
  “distributed”,
  “data”,
  “processing”
]
```

```
def retrieve(item_idx):
    time.sleep(item_idx / 10.)
    return item_idx, database[item_idx]

@ray.remote
def retrieve_task(item_idx):
    return retrieve(item_idx)
```

```
obj_refs = [
        retrieve_task.remote(idx) for idx in range(6)
]
data = ray.get(obj_refs)
```

Q: How would driver share data with distributed workers?



Cluster of machines

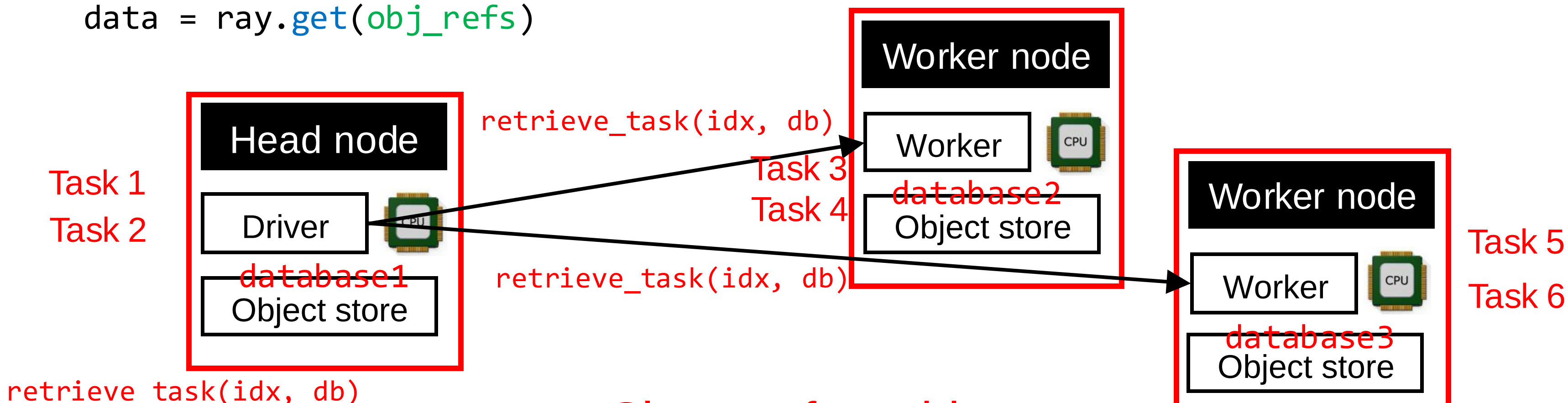
Ray API: Distributed object store

```
database = [
  “learning”,
  “Ray”,
  “for”,
  “distributed”,
  “data”,
  “processing”
]
```

```
@ray.remote
def retrieve_task(item_idx, db):
    time.sleep(item_idx / 10.)
    return item_idx, db[item_idx]
```

Use distributed object store to share data across all workers in the cluster

```
db_obj_ref = ray.put(database)
obj_refs = [retrieve_task.remote(idx, db_obj_ref) for idx in range(6)]
data = ray.get(obj_refs)
```



Cluster of machines

Ray API: Actors

Ray actor class definition

```
@ray.remote
class DataTracker:
    def __init__(self):
        self._counts = 0
    def increment(self):
        self._counts += 1
    def counts(self):
        return self._counts
```

```
database = [
  "learning",
  "Ray",
  "for",
  "distributed",
  "data",
  "processing"
]
```

Ray task definition

```
@ray.remote
def retrieve_task_n_track(item_idx, tracker, db):
    time.sleep(item_idx / 10.)
    tracker.increment.remote()
    return item_idx, db[item_idx]
```

Invoke remote actor

```
tracker = DataTracker.remote()
obj_refs = [
       retrieve_task_n_track.remote(idx, tracker, db_obj_ref) for idx in range(6)
]
data = ray.get(obj_refs)
print(ray.get(tracker.counts.remote()))
```

Invoke remote tasks

Ray tasks are decorated Python functions.

Ray **actors** are **decorated Python classes,** which encapsulate **state**.

Actors allows you to run **stateful** computations on a cluster.

Demo ...

An example task graph