- Ray is a distributed system specifically built for reinforcement learning
    - It has since emerged into a much more general-purpose platform, but we will limit the discussion to its support for RL

- What is supervised learning?
    - Inputs have labels, and the model is trained on the input+label pairs
    - The model is a deep neural network
- What is reinforcement learning?
    - “not only to exploit the data gathered, but also to explore the space of possible actions”
    - “RL deals with learning to operate continuously within an uncertain environment based on delayed and limited feedback”
    - “The central goal of an RL application is to learn a policy—a mapping from the state of the environment to a choice of action—that yields effective performance over time, e.g., winning a game or piloting a drone.”

- What does RL require?
    - 1) **Simulation** to evaluate policies and explore different actions
    - 2) **Distributed training** to improve the policy based on the data produced from the simulations
    - 3) **Serve** the policy
        - Closed loop = you make choices and see how they do
        - Open loop = you just make choices, but you don’t see how they do
- How does RL work?
    - Agent interacts with environment
    - Agent wants to learn policy to maximize a reward
    - Policy is a mapping from the **state of an environment** to an **action**

    - Two-step process:
        - **Policy evaluation**
        - **Policy improvement**

        - Policy evaluation:
            - Agent interacts with environment and generates a **trajectory**
                - i.e., a sequence of (state, reward) tuples produced by the current policy
        - Policy improvement:
            - Agent uses trajectories to improve policy in the direction that maximizes the reward
- There are already existing solutions that do all three of those things
    - Why not just combine these existing solutions?
    - Why do we need a whole new system?
    - **There is a tight coupling between all three of these requirements, and stitching together existing systems will not yield good performance**

    - **Furthermore, in the context of RL, application logic tends to be tightly coupled with the underlying system, so this makes it even more difficult to stitch multiple systems together**

- What does Ray provide?
    - Support for fine-grained computations
    - Support for heterogeneity
        - Simulations take different amounts of time (milliseconds vs. hours)
        - Resources (CPUs, GPUs, TPUs, and so on)
    - Flexible computation model
        - Stateless and stateful computations
            - Ray can express both:
                - Task-parallel computations
                - Actor-based computations
                - **What is the difference between the two? Why do you need both instead of just one of them?**
    - Dynamic execution
        - We don't know what order things will finish in or even which tasks will be invoked through the application lifetime
    - Support for millions of tasks per second
    - Integrates nicely with existing simulators and deep learning frameworks

- Existing systems support these forms of compute, such as MapReduce, Spark, etc.
    - Why not just use these?
    - No support for **serving** or **fine-grained simulations**
    - But in turn, these systems have a more expansive API and functionality than Ray, so they still serve an important purpose

    - I kind of view Ray as a CPU and bulk-synchronous parallel systems as a GPU

Programming and Computation Model

- Models an application as a graph of dependent tasks
- Tasks
    - Remote function on a *stateless* worker
    - Returns a future that can be dereferenced to get the result
    - Tasks operate on **immutable** functions
    - Thus, tasks are idempotent, which significantly simplifies fault tolerance, as we saw on Monday
        - Just re-execute the tasks!
- Actors
    - Similar to a class in a program
    - *Stateful* computation
- What are the pros and cons of each?

- See Table 2 in the paper

- Ray API (see Table 1)

- Ray represents an application with a computation graph
- Two kinds of nodes:
  - **Data objects**
  - **Remote function invocations**
- Three kinds of edges:
  - Data edges
    - Capture dependencies between objects and tasks
  - Control edges
    - Capture computation dependencies that result from nested remote functions
  - Stateful edges
    - Captures state dependencies between multiple method invocations on the same actor
    - Useful for:
      - (1) Capturing implicit data dependencies on the internal actor state between successive invocations of the actor
      - (2) Maintaining lineage

Architecture

- Contains:
  - An **application layer** to implement the API
  - A **system layer** providing high scalability and fault tolerance
    - This layer is responsible for tracking the status of futures, as we discussed on Monday

- Application layer
  - Has the following components:
    - A driver (process that executes the program)
    - Worker (executes stateless functions)
    - Actor (a stateful process)

- System layer
  - Global Control Store (GCS)
    - Maintains control state of the system, such as where objects are located and what their sizes are
    - GCS is a key-value store backed by Redis
    - Achieves scale with sharding
    - Provides fault tolerance with per-shard chain replication

        - Importantly, GCS stores object metadata that other systems store in teh scheduler.
            - This is important for providing low latency and high throughput performance
                - The GCS and the scheduler can scale independently, and the scheduler is not on the critical path of task dispatch (which requires asking where the objects are and what their sizes are before launching a task)
    - Bottom-up Distributed Scheduler
        - Two-level hierarchical scheduler
            - Global scheduler
            - Per-node local schedulers
        - Submitted tasks are first sent to the local scheduler
        - If no local resources are available, the global scheduler is involved
            - Maybe the machine is overloaded, or lacks a resource such as a GPU
            - By going to the local scheduler first, we can prevent the global scheduler from becoming a bottleneck
        - The global scheduler chooses a machine that can (1) provide the required resources and (2) has the lowest *estimated waiting time*
            - Estimated waiting time = (1) estimated queuing time of task at node + (2) estimated transfer time of task's remote inputs
                - **I'm not sure why the estimated transfer time of the inputs is considered. Presumably this could be overlapped with the queuing time.**
                - **In order words, it seems like the estimated waiting time should be:**
                  max(estimated queuing time, estimated transfer time)
    - In-Memory Distributed Object Store
        - Transfer objects on the same node via **shared memory**
            - This allows **zero-copy**
        - Replicate objects across nodes so that tasks/actors on remote nodes have local access to objects
        - Objects are immutable, which significantly simplifies the consistency protocol and fault tolerance
        - On failure, objects are recovered through lineage re-execution (as discussed on Monday)
        - GCS stores the lineage for both tasks (stateless) and actors (stateful)
        - Only support objects that can fit on a single node (i.e., large objects such as large matrices and trees require support at the application level… Ray does not provide native support for these)