Ray (Anyscale)

"Ray seeks to enable the development and composition of distributed applications and libraries *in general*"

Simulation + Training + Serving for RL applications

Bulk-Synchronous parallel

Map-Reduce, Spark, Dryad

No fine-grain simulation or policy serving

Distributed DL

TFlow, MXNet

No default support for simulation and serving

Streaming

Naiad, Storm

Task-Parallel

CIEL, Dask

Little support for distributed training and serving

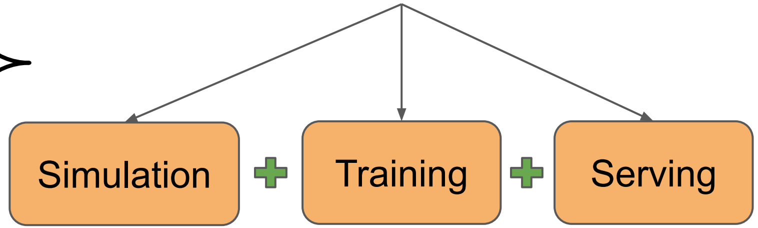
Model-Serving

TFlow Serving, Clipper

No training and simulation

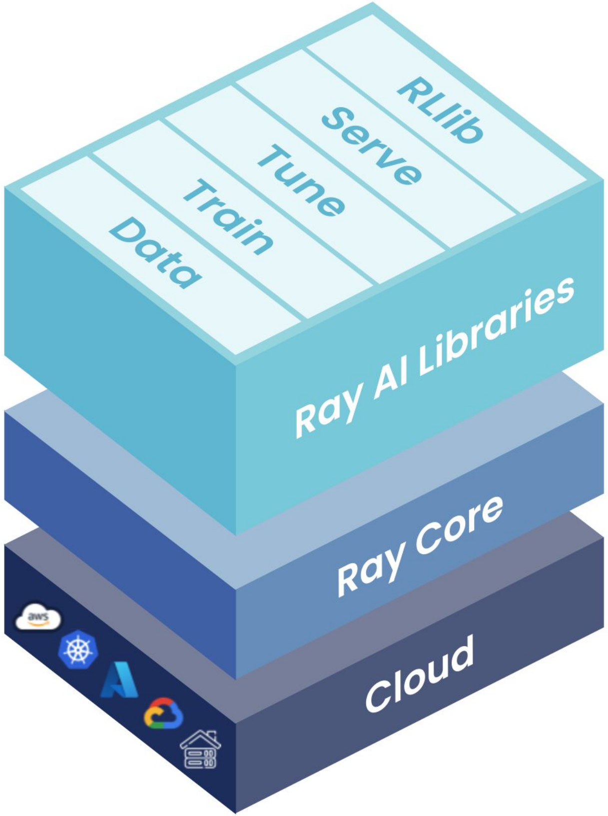
Ray Framework

- General-Purpose
- Cluster Computing



Created at RISELab, UC Berkeley (also Apache Sparks, Databricks) – Pr. Ion Stoica

Ray Layers



High-level libraries that enable scaling of AI workloads

Low-level distributed computing framework with concise C++ core and simple Python API

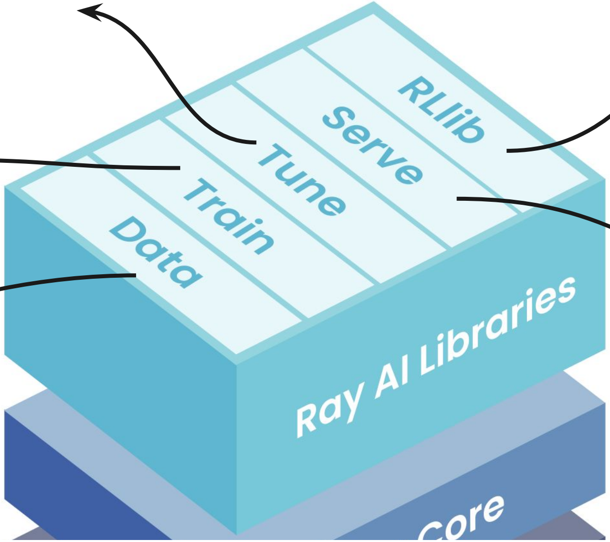
Integration with cluster environments for cloud deployment

Ray AI Libraries

Hyper-parameter
tuning at scale

High-Performance
RL library

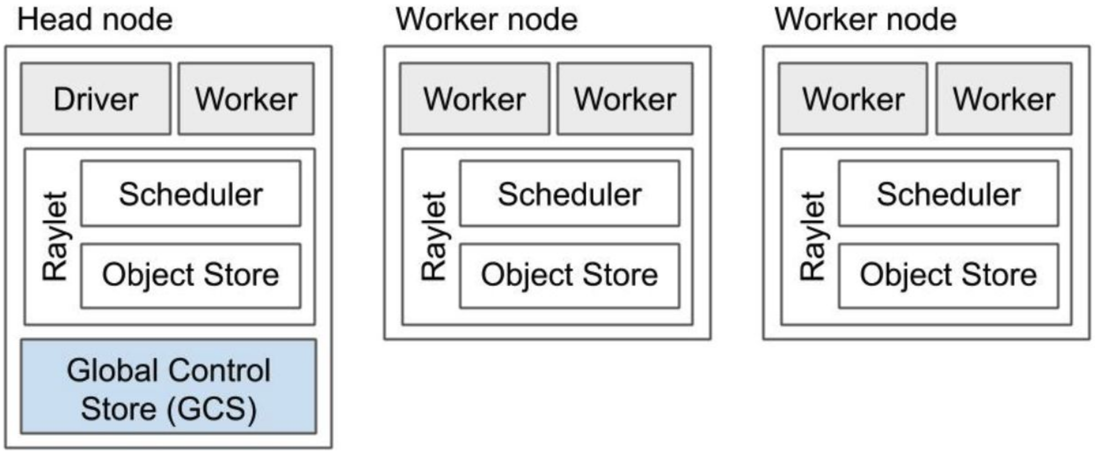
Distributed model
training



Scalable and
programmable
model serving

Distributed data
processing

Ray Cluster



Separate worker nodes are instantiated in multiple-server settings.
In single-server settings Ray creates multiple **Worker** processes according to available CPUs

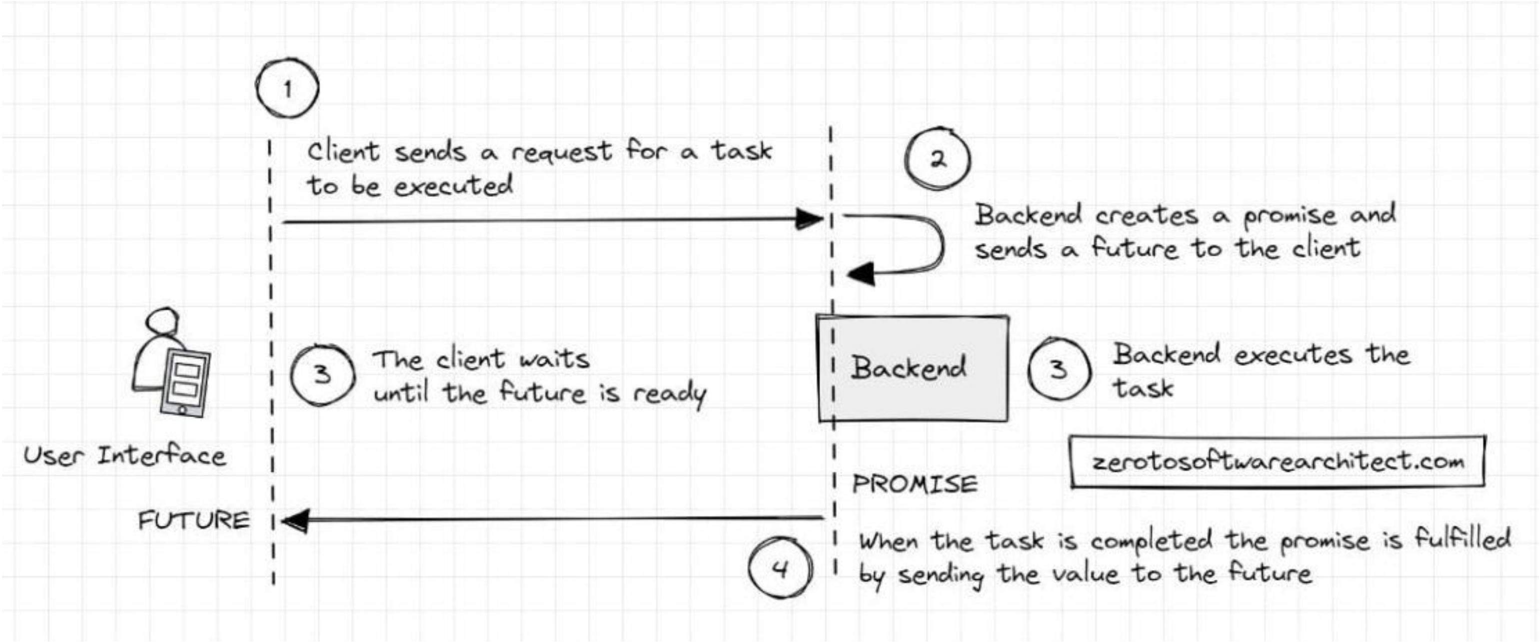
Node that calls **ray.init()**
GCS:

- manages cluster-level metadata, such as the locations of actors, stored as key-value pairs that may be cached locally by workers.
- manages cluster-level operations like scheduling for placement groups, actors, cluster-node membership

Raylet:

- Scheduler responsible for resource management, task placement, and fulfilling task arguments
- Shared-memory object store (Plasma Object Store) that stores, transfers and spills large objects

Futures & Promises



['Futures and Promises', scaleyourapp.com]

Simple Python API with Decorators

```python
import ray
from random import randint

number = 12

@ray.remote
def create_policy():
    return randint(1, 20)

@ray.remote
def update_policy(policy, rollout):
    if rollout == 1:
        return policy
    else:
        if rollout == 0:
            return policy + 1
        elif rollout == 2:
            return policy - 1
```

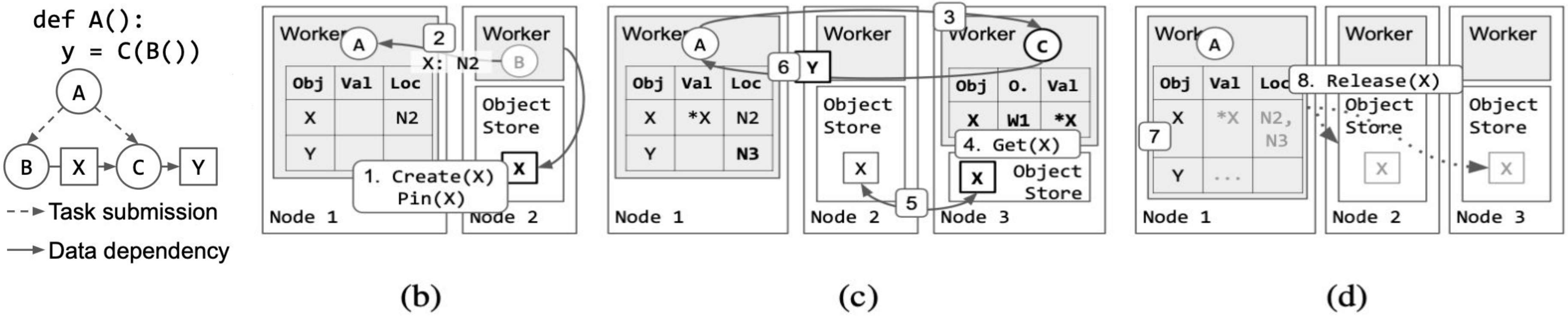
```python
@ray.remote
class Simulator(object):
    def __init__(self):
        # Initialize environment
        self.env = number

    def rollout(self, policy):
        # policy-based action
        action = policy
        # simulate environment
        if action < self.env:
            return 0
        elif action == self.env:
            return 1
        elif action > self.env:
            return 2

@ray.remote
def train_policy():
    # Create a policy
    policy = create_policy.remote()
    # Create actor
    simulator = Simulator.remote()
    # Do 5 steps of training
    for _ in range(3):
        # Rollout on actor
        rollout = simulator.rollout.remote(policy)
        # Update policy
        policy = update_policy.remote(policy, rollout)

    return ray.get(policy)
```

Ownership Model



1. Large objects created and pinned at remote node (first-primary copy, deleted last)

2. Returned by reference to owner (Node 1)

3. Request of small object

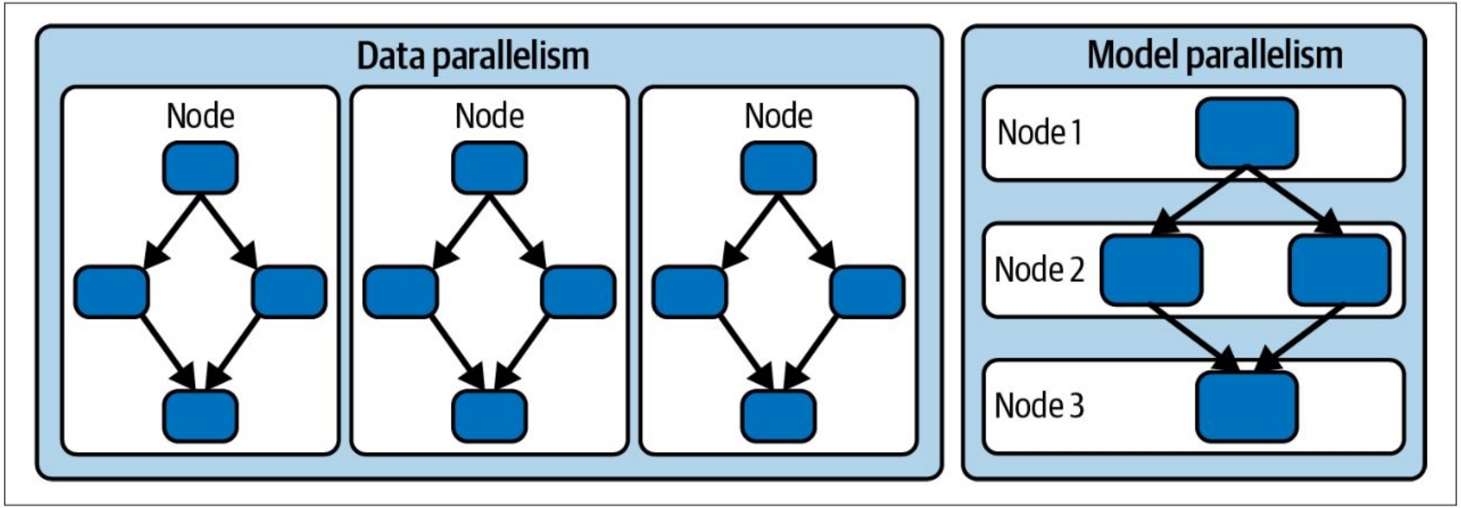
4-5. Get(X) call to dereference large object (cached)

6. Small objects are returned by value

7. Value deletion from local table

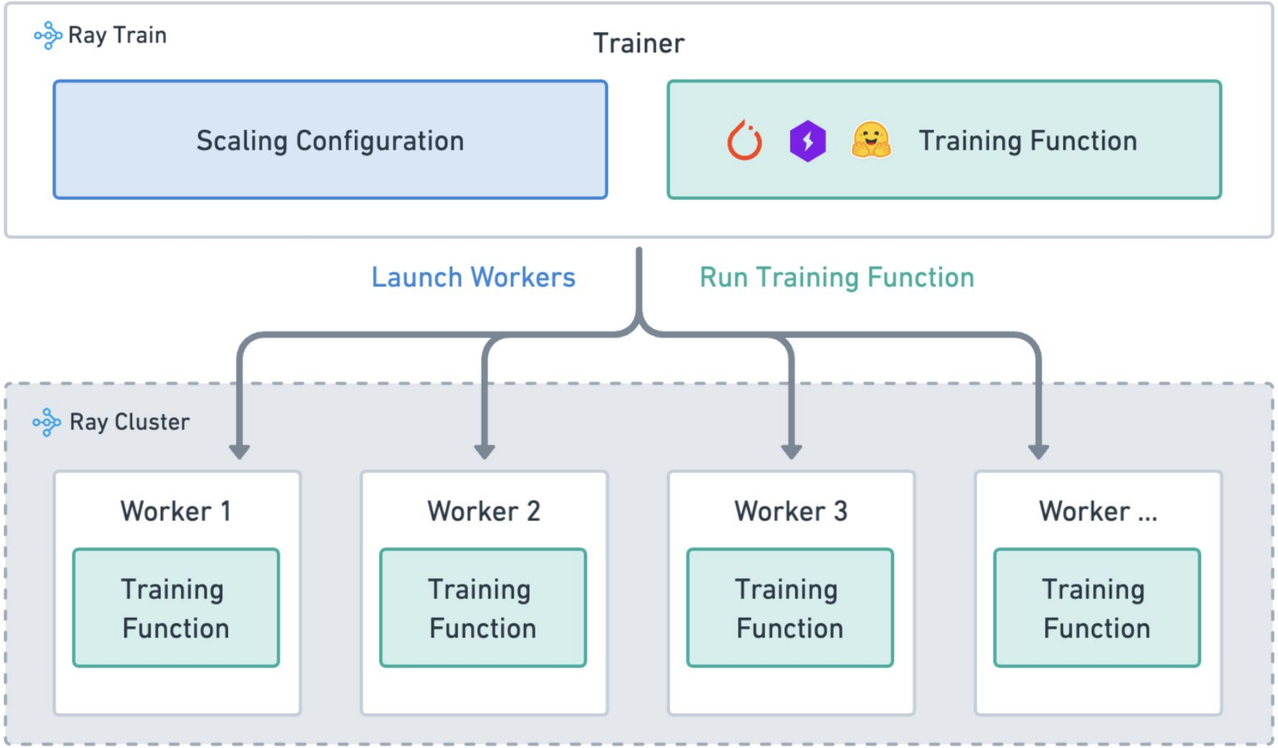
8. Owner releases the large object

Distributed Training



Ray Train is designed for distributed data-parallel training

Data-Parallel Ray Workflow



Training Function

Load model/dataset, train, save checkpoints and store metrics

Data-Parallel Training Example – Preprocessing

```
def get_dataloaders(batch_size):
    # Transform to normalize the input images.
    transform = transforms.Compose([ToTensor(), Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])

    with FileLock(os.path.expanduser("~/data.lock")):
        # Download training data from open datasets.
        training_data = datasets.CIFAR10(
            root="~/data",
            train=True,
            download=True,
            transform=transform,
        )

        # Download test data from open datasets.
        testing_data = datasets.CIFAR10(
            root="~/data",
            train=False,
            download=True,
            transform=transform,
        )

    # Create data loaders.
    train_dataloader = DataLoader(training_data, batch_size=batch_size, shuffle=True)
    test_dataloader = DataLoader(testing_data, batch_size=batch_size)

    return train_dataloader, test_dataloader
```

Data-Parallel Training Example – Train Function

```python
def train_func_per_worker(config):
    lr = config["lr"]
    epochs = config["epochs"]
    batch_size = config["batch_size_per_worker"]

    # Get data loaders inside the worker training function.
    train_dataloader, valid_dataloader = get_dataloaders(batch_size=batch_size)

    # [1] Prepare data loader for distributed training.
    # The prepare_data_loader method assigns unique rows of data to each worker so that
    # the model sees each row once per epoch.
    # NOTE: This approach only works for map-style datasets. For a general distributed
    # preprocessing and sharding solution, see the next part using Ray Data for data
    # ingestion.
    # =================================================================================
    train_dataloader = ray.train.torch.prepare_data_loader(train_dataloader)
    valid_dataloader = ray.train.torch.prepare_data_loader(valid_dataloader)

    model = VisionTransformer(
        image_size=32,   # CIFAR-10 image size is 32x32
        patch_size=4,    # Patch size is 4x4
        num_layers=12,   # Number of transformer layers
        num_heads=8,     # Number of attention heads
        hidden_dim=384,  # Hidden size (can be adjusted)
        mlp_dim=768,     # MLP dimension (can be adjusted)
        num_classes=10   # CIFAR-10 has 10 classes
    )
```

```python
    # [2] Prepare and wrap your model with DistributedDataParallel.
    # The prepare_model method moves the model to the correct GPU/CPU device.
    # =======================================================================
    model = ray.train.torch.prepare_model(model)

    loss_fn = nn.CrossEntropyLoss()
    optimizer = torch.optim.AdamW(model.parameters(), lr=lr, weight_decay=1e-2)

    # Model training loop.
    for epoch in range(epochs):
        if ray.train.get_context().get_world_size() > 1:
            # Required for the distributed sampler to shuffle properly across epochs.
            train_dataloader.sampler.set_epoch(epoch)

        model.train()
        for X, y in tqdm(train_dataloader, desc=f"Train Epoch {epoch}"):
            pred = model(X)
            loss = loss_fn(pred, y)

            optimizer.zero_grad()
            loss.backward()
            optimizer.step()

        model.eval()
        valid_loss, num_correct, num_total = 0, 0, 0
        with torch.no_grad():
            for X, y in tqdm(valid_dataloader, desc=f"Valid Epoch {epoch}"):
                pred = model(X)
                loss = loss_fn(pred, y)

                valid_loss += loss.item()
                num_total += y.shape[0]
                num_correct += (pred.argmax(1) == y).sum().item()

        valid_loss /= len(train_dataloader)
        accuracy = num_correct / num_total
```

Data-Parallel Training Example – Train Function

```python
# [3] (Optional) Report checkpoints and attached metrics to Ray Train.
# ==============================================================
with tempfile.TemporaryDirectory() as temp_checkpoint_dir:
    torch.save(
        model.module.state_dict(),
        os.path.join(temp_checkpoint_dir, "model.pt")
    )
    ray.train.report(
        metrics={"loss": valid_loss, "accuracy": accuracy},
        checkpoint=ray.train.Checkpoint.from_directory(temp_checkpoint_dir),
    )
    if ray.train.get_context().get_world_rank() == 0:
        print({"epoch_num": epoch, "loss": valid_loss, "accuracy": accuracy})
```

Data-Parallel Training Example – Configuration & Results

```python
def train_cifar_10(num_workers, use_gpu):
    global_batch_size = 512

    train_config = {
        "lr": 1e-3,
        "epochs": 1,
        "batch_size_per_worker": global_batch_size // num_workers,
    }

    # [1] Start distributed training.
    # Define computation resources for workers.
    # Run `train_func_per_worker` on those workers.
    # =============================================
    scaling_config = ScalingConfig(num_workers=num_workers, use_gpu=use_gpu)
    run_config = RunConfig(
        name=f"train_run-{uuid.uuid4().hex}",
    )
    trainer = TorchTrainer(
        train_loop_per_worker=train_func_per_worker,
        train_loop_config=train_config,
        scaling_config=scaling_config,
        run_config=run_config,
    )
    result = trainer.fit()
    print(f"Training result: {result}")

if __name__ == "__main__":
    train_cifar_10(num_workers=4, use_gpu=True)
```


```json
{
    "loss": 0.35899272256967973,
    "accuracy": 0.3228,
    "timestamp": 1758600479,
    "checkpoint_dir_name": "checkpoint_000000",
    "should_checkpoint": true,
    "done": false,
    "training_iteration": 1,
    "trial_id": "d6250_00000",
    "date": "2025-09-23_00-07-59",
    "time_this_iter_s": 21.89649200439453,
    "time_total_s": 21.89649200439453,
    "pid": 2409569,
    "hostname": "yecl-gpu-cluster",
    "node_ip": "192.168.0.121",
    "config": {
        "train_loop_config": {
            "lr": 0.001,
            "epochs": 1,
            "batch_size_per_worker": 128
        }
    },
    "time_since_restore": 21.89649200439453,
    "iterations_since_restore": 1
}
```

DeepSpeed Library

“Deep learning optimization software suite that powers unprecedented scale and speed for both training and inference”

Training
- Speed Scale Cost
- Democratization
- MoE models
- Long sequence
- RLHF

Inference
- Large models
- Latency
- Serving cost
- Agility

Compression
- Model size
- Latency
- Composability
- Runnable on client devices

Science
- Speed
- Scale
- Capability
- Diversity
- Discovery

Zero Redundancy Optimizer (ZeRO): Sharding model states, gradients and optimizer states across GPU to reduce memory usage

ZeRO-3 Training Example – Train Function

```python
def train_func(config):
    """Your training function that will be launched on each worker."""

    # Unpack training configs
    set_seed(config["seed"])
    num_epochs = config["num_epochs"]
    train_batch_size = config["train_batch_size"]
    eval_batch_size = config["eval_batch_size"]

    # Instantiate the Model
    model = AutoModelForSequenceClassification.from_pretrained(
        "bert-base-cased", return_dict=True
    )

    # Prepare Ray Data Loaders
    # ====================================================
    train_ds = ray.train.get_dataset_shard("train")
    eval_ds = ray.train.get_dataset_shard("validation")

    tokenizer = AutoTokenizer.from_pretrained("bert-base-cased")

    def collate_fn(batch):
        outputs = tokenizer(
            list(batch["sentence1"]),
            list(batch["sentence2"]),
            truncation=True,
            padding="longest",
            return_tensors="pt",
        )
        outputs["labels"] = torch.LongTensor(batch["label"])
        return outputs
```

```python
    train_dataloader = train_ds.iter_torch_batches(
        batch_size=train_batch_size, collate_fn=collate_fn
    )
    eval_dataloader = eval_ds.iter_torch_batches(
        batch_size=eval_batch_size, collate_fn=collate_fn
    )
    # ====================================================

    # Initialize DeepSpeed Engine
    model, optimizer, _, lr_scheduler = deepspeed.initialize(
        model=model,
        model_parameters=model.parameters(),
        config=deepspeed_config,
    )
    device = get_accelerator().device_name(model.local_rank)

    # Initialize Evaluation Metrics
    f1 = BinaryF1Score().to(device)
    accuracy = BinaryAccuracy().to(device)
```

ZeRO-3 Training Example – Train Function

```python
    for epoch in range(num_epochs):
        # Training
        model.train()
        for batch in train_dataloader:
            batch = {k: v.to(device) for k, v in batch.items()}
            outputs = model(**batch)
            loss = outputs.loss
            model.backward(loss)
            optimizer.step()
            lr_scheduler.step()
            optimizer.zero_grad()

        # Evaluation
        model.eval()
        for batch in eval_dataloader:
            batch = {k: v.to(device) for k, v in batch.items()}
            with torch.no_grad():
                outputs = model(**batch)
            predictions = outputs.logits.argmax(dim=-1)

            f1.update(predictions, batch["labels"])
            accuracy.update(predictions, batch["labels"])

        # torchmetrics will aggregate the metrics across all workers
        eval_metric = {
            "f1": f1.compute().item(),
            "accuracy": accuracy.compute().item(),
        }
        f1.reset()
        accuracy.reset()
```

```python
        if model.global_rank == 0:
            print(f"epoch {epoch}:", eval_metric)

        # Report checkpoint and metrics to Ray Train
        # ==============================================================
        with TemporaryDirectory() as tmpdir:
            # Each worker saves its own checkpoint shard
            model.save_checkpoint(tmpdir)

            # Ensure all workers finished saving their checkpoint shard
            torch.distributed.barrier()

            # Report checkpoint shards from each worker in parallel
            ray.train.report(
                metrics=eval_metric, checkpoint=Checkpoint.from_directory(tmpdir)
            )
        # ==============================================================
```

ZeRO-3 Training Example – Configuration

```python
if __name__ == "__main__":
    deepspeed_config = {
        "optimizer": {
            "type": "AdamW",
            "params": {
                "lr": 2e-5,
            },
        },
        "scheduler": {"type": "WarmupLR", "params": {"warmup_num_steps": 100}},
        "fp16": {"enabled": True},
        "bf16": {"enabled": False},  # Turn this on if using AMPERE GPUs.
        "zero_optimization": {
            "stage": 3,
            "offload_optimizer": {
                "device": "none",
            },
            "offload_param": {
                "device": "none",
            },
        },
        "gradient_accumulation_steps": 1,
        "gradient_clipping": True,
        "steps_per_print": 10,
        "train_micro_batch_size_per_gpu": 16,
        "wall_clock_breakdown": False,
    }
```

```python
    training_config = {
        "seed": 42,
        "num_epochs": 3,
        "train_batch_size": 16,
        "eval_batch_size": 32,
        "deepspeed_config": deepspeed_config,
    }

    # Prepare Ray Datasets
    hf_datasets = load_dataset("glue", "mrpc")

    # Convert HuggingFace datasets to Ray datasets using from_items
    ray_datasets = {
        "train": ray.data.from_items([dict(item) for item in hf_datasets["train"]]),
        "validation": ray.data.from_items([dict(item) for item in hf_datasets["validation"]]),
    }

    trainer = TorchTrainer(
        train_func,
        train_loop_config=training_config,
        scaling_config=ScalingConfig(num_workers=4, use_gpu=True),
        datasets=ray_datasets,
        dataset_config=DataConfig(datasets_to_split=["train", "validation"]),
    )

    result = trainer.fit()
```

ZeRO-3 Training Example – Results


```
{
    "f1": 0.8944723606109619,
    "accuracy": 0.845588207244873,
    "timestamp": 1758598124,
    "checkpoint_dir_name": "checkpoint_000002",
    "should_checkpoint": true,
    "done": false,
    "training_iteration": 3,
    "trial_id": "3a66a_00000",
    "date": "2025-09-22_23-28-44",
    "time_this_iter_s": 12.80349349975586,
    "time_total_s": 73.79229402542114,
    "pid": 2378503,
    "hostname": "yecl-gpu-cluster",
    "node_ip": "192.168.0.121",
```

```
"config": {
    "train_loop_config": {
        "seed": 42,
        "num_epochs": 3,
        "train_batch_size": 16,
        "eval_batch_size": 32,
        "deepspeed_config": {
            "optimizer": {
                "type": "AdamW",
                "params": {
                    "lr": 2e-05
                }
            },
            "scheduler": {
                "type": "WarmupLR",
                "params": {
                    "warmup_num_steps": 100
                }
            },
            "fp16": {
                "enabled": true
            },
            "bf16": {
                "enabled": false
            },
```

```
            "zero_optimization": {
                "stage": 3,
                "offload_optimizer": {
                    "device": "none"
                },
                "offload_param": {
                    "device": "none"
                }
            },
            "gradient_accumulation_steps": 1,
            "gradient_clipping": true,
            "steps_per_print": 10,
            "train_micro_batch_size_per_gpu": 16,
            "wall_clock_breakdown": false
        }
    }
},
"time_since_restore": 73.79229402542114,
"iterations_since_restore": 3
}
```

Model-Parallel Training – Preparing Model

```python
# -----------------------------
# 1. Define pipeline-parallel model
# -----------------------------
def create_pipeline_model(num_stages=2):
    """Create a pipeline model with proper layer specifications"""
    # Split the model into a sequence of layers
    layers = [
        LayerSpec(nn.Linear, 128, 256),    # Stage 0
        LayerSpec(nn.ReLU),                # Stage 0
        LayerSpec(nn.Linear, 256, 512),    # Stage 0
        LayerSpec(nn.ReLU),                # Stage 0
        LayerSpec(nn.Linear, 512, 256),    # Stage 1
        LayerSpec(nn.ReLU),                # Stage 1
        LayerSpec(nn.Linear, 256, 128),    # Stage 1
        LayerSpec(nn.ReLU),                # Stage 1
        LayerSpec(nn.Linear, 128, 10),     # Stage 1 (output layer)
    ]

    # Pipeline with specified number of stages
    model = PipelineModule(
        layers=layers,
        num_stages=num_stages,
        loss_fn=nn.CrossEntropyLoss(),
        partition_method="uniform",  # split layers evenly across stages
        activation_checkpoint_interval=0,  # disable for simplicity
    )
    return model
```

Utilize Deepspeed's *PipelineModule* to split the model into **num_stages** partitions

Model-Parallel Training – Train Function

```python
# -----------------------------
# 2. Ray Train worker function
# -----------------------------
def train_func(config):
    """Training function for pipeline parallel model"""
    # Get Ray Train context (Ray handles distributed initialization)
    context = train.get_context()
    world_rank = context.get_world_rank()
    world_size = context.get_world_size()
    local_rank = context.get_local_rank()

    print(f"Worker {world_rank}/{world_size}, Local rank: {local_rank}")

    # Set device
    device = f"cuda:{local_rank}" if torch.cuda.is_available() else "cpu"
    torch.cuda.set_device(local_rank)

    # Initialize DeepSpeed distributed backend explicitly for PipelineModule
    if not dist.is_initialized():
        # Initialize with the same backend Ray Train uses
        dist.init_process_group(
            backend="nccl" if torch.cuda.is_available() else "gloo",
            world_size=world_size,
            rank=world_rank
        )
        print(f"Initialized distributed backend on rank {world_rank}")

```

```python
    # Initialize DeepSpeed's distributed backend
    import deepspeed.comm as dist_comm
    if not dist_comm.is_initialized():
        dist_comm.init_distributed(
            dist_backend="nccl" if torch.cuda.is_available() else "gloo",
            auto_mpi_discovery=False,
            distributed_port=29500,
            verbose=False
        )
        print(f"DeepSpeed distributed backend initialized on rank {world_rank}")

    # Create pipeline model
    num_stages = config.get("num_stages", 2)
    model = create_pipeline_model(num_stages=num_stages)

    print(f"Created pipeline model with {num_stages} stages")

    # Create proper DeepSpeed configuration with batch size constraints
    deepspeed_config = config["deepspeed_config"].copy()
```

Model-Parallel Training – Train Function

```python
    deepspeed_config.update({
        "train_batch_size": train_batch_size,
        "train_micro_batch_size_per_gpu": micro_batch_size,
        "gradient_accumulation_steps": gradient_accumulation_steps,
    })

    # Initialize DeepSpeed with pipeline model
    engine, optimizer, _, _ = deepspeed.initialize(
        model=model,
        config=deepspeed_config,
        dist_init_required=False,  # Ray Train handles distributed init
    )
```

Model-Parallel Training – Configuration

```python
if __name__ == "__main__":
    ray.init(num_gpus=2)
    print("Ray initialized for pipeline-parallel training")

    # DeepSpeed config for pipeline parallelism + ZeRO stage 1
    deepspeed_config = {
        "train_micro_batch_size_per_gpu": 4,  # Smaller batch size per GPU
        "gradient_accumulation_steps": 2,     # Accumulate gradients
        "optimizer": {
            "type": "Adam",
            "params": {"lr": 1e-3, "weight_decay": 0.01},
        },
        "fp16": {"enabled": False},  # Keep FP32 for simplicity and stability
        "zero_optimization": {
            "stage": 1,  # Stage 1 works well with pipeline parallelism
            "allgather_partitions": True,
            "reduce_scatter": True,
            "allgather_bucket_size": 2e8,
            "reduce_bucket_size": 2e8,
        },
        "steps_per_print": 5,
        "wall_clock_breakdown": False,

        # Pipeline-specific configurations
        "pipeline": {
            "pipe_partitioned": True,
            "grad_partitioned": True,
        }
    }
```

```python
    # Training configuration
    train_config = {
        "epochs": 2,  # Reduced epochs for demo
        "num_stages": 2,  # Number of pipeline stages (should match num_workers)
        "deepspeed_config": deepspeed_config,
    }

    # Scaling configuration - one worker per pipeline stage
    scaling_config = ScalingConfig(
        num_workers=2,   # Must match num_stages for pipeline parallelism
        use_gpu=True,
        resources_per_worker={"GPU": 1},  # One GPU per worker/stage
    )

    # Run configuration
    run_config = RunConfig(
        # verbose=2,  # Enable for more detailed logging
    )

    # Create trainer
    trainer = TorchTrainer(
        train_func,
        train_loop_config=train_config,
        scaling_config=scaling_config,
        run_config=run_config,
    )
```

Model-Parallel Training – Results

```python
try:
    result = trainer.fit()
    print("\nPipeline-parallel training completed successfully!")
    print(f"Final metrics: {result.metrics}")

except Exception as e:
    print(f"Training failed with error: {e}")
    print("Common issues:")
    print("1. Make sure you have at least 2 GPUs available")
    print("2. Ensure DeepSpeed is properly installed")
    print("3. Check that CUDA and NCCL are working correctly")

finally:
    ray.shutdown()
    print("Ray shut down")
```


```json
{
    "loss": 2.2838621139526367,
    "epoch": 1,
    "learning_rate": 0.001,
    "timestamp": 1758667846,
    "checkpoint_dir_name": "None",
    "done": true,
    "training_iteration": 2,
    "trial_id": "b853e_00000",
    "date": "2025-09-23_18-50-46",
    "time_this_iter_s": 0.2586488723754883,
    "time_total_s": 6.3532164096832275,
    "pid": 2492297,
    "hostname": "yecl-gpu-cluster",
    "node_ip": "192.168.0.121",
    "config": {
        "train_loop_config": {
            "epochs": 2,
            "num_stages": 2,
            "deepspeed_config": {
                "train_micro_batch_size_per_gpu": 4,
                "gradient_accumulation_steps": 2,
                "optimizer": {
                    "type": "Adam",
                    "params": {
                        "lr": 0.001,
                        "weight_decay": 0.01
```

Resources

- Ray Documentation
- Data-Parallel Pytorch Example
- ZeRO-3 DeepSpeed Example
- Ray Paper (2017)
- Ownership Paper (2021)
- Ray 2.0 Architecture Whitepaper
- DeepSpeed
- OReilly - Learning Ray: Flexible Distributed Python for Machine Learning