Integrating Cerebro with Ray

Abhishek Gupta
University of California San Diego
La Jolla, California, USA

Rishikesh Ingale
University of California San Diego
La Jolla, California, USA

ABSTRACT

Cerebro is a model selection system that introduced the concept of model hopper parallelism (MOP), which is a strategy that inherits the upsides of data parallelism and task parallelism but not their downsides. Cerebro allows for the addition of new execution backends, which can be done by implementing a new class that extends its abstract Backend class. Using Ray, an API for building distributed applications in Python, we have created an additional backend that performs Model Hopper Parallelism for scalable and efficient model selection on deep learning models created with standard Python machine learning libraries. Our work includes sharding the processed training data across multiple machines and implementing MOP over the different models on these shards.

KEYWORDS

model hopper parallelism, model selection, deep learning, machine learning, Ray

ACM Reference Format:
Abhishek Gupta and Rishikesh Ingale. 2021. Integrating Cerebro with Ray. In *Proceedings of ACM Conference (Conference'21).* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Deep learning has made many strides in the fields of computer vision, natural language processing, and reinforcement learning to name a few. Such advances have been made possible due to the ever-increasing number of parameters within the models used as well as the introduction of novel architectures. However, as models get larger and more complex, the problem of model selection becomes more difficult. More often than not, ML practitioners want to experiment with not only the number of parameters but also different architectures as well as hyperparameters in order to alleviate the effects of underfitting or overfitting. This presents a massive bottleneck for the adoption of deep learning by enterprises and domain scientists.

To address this, Nakandala et al. created Cerebro [4], a data system for model selection which implements a new type of parallelism called Model Hopper Parallelism (MOP), a hybrid of data and task parallelism. The system is made open source to allow the addition of new execution backends. Currently, MOP is implemented in Cerebro using a Spark backend.

We were tasked with implementing a backend using the Ray library. Ray is an API that allows users to build distributed applications in Python. It offers various primitives that can be leveraged in order to implement remote procedural calls (RPC), specify resources, and shard data. This was all needed in order to replace Spark dependencies found in the original backend. The results obtained from the experiments demonstrate similar performance to that of the Spark backend.

2 BACKGROUND

In this section, we provide some background on how MOP works at a high level, the structure of the Cerebro system, and the specific primitives that the Ray API offers and have been of use in this project.

2.1 Cerebro

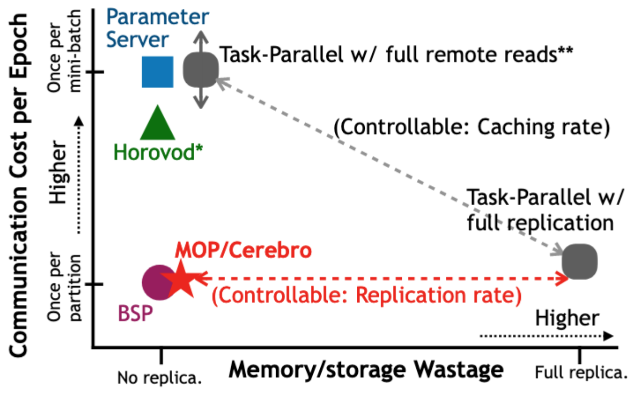

Figure 1: MOP Performance [4]

Cerebro distributes training across a cluster of workers. Assuming that it is given S configurations to train on the same dataset and $p < S$ workers, Cerebro starts by splitting the data into p shards, and places one on each worker. When the time comes to execute a single training epoch for all the configurations, Cerebro starts by creating a set of worker-configuration pairs and shuffling them randomly. Depending on the scheduling algorithm as well as the idleness of the workers and configurations (the latter being defined as currently training a configuration and the former being defined as currently being trained on a worker), Cerebro makes the configurations visit the workers such that each configuration visits all workers only once during the current epoch. When a configuration visits a worker, the worker trains the configuration on its shard using one pass of stochastic gradient descent (SGD); this is called a sub-epoch. When a sub-epoch is completed, both the worker and configuration become idle. This allows Cerebro to remove that particular pair from the set of worker-configuration pairs and schedule sub-epochs for other pairs. This property of the configurations "hopping" from worker to worker is what gives MOP its name. As one can deduce from this description of MOP, it inherits the benefits of data parallelism and task parallelism without incurring their memory, storage, and communication costs (see Figure 1).

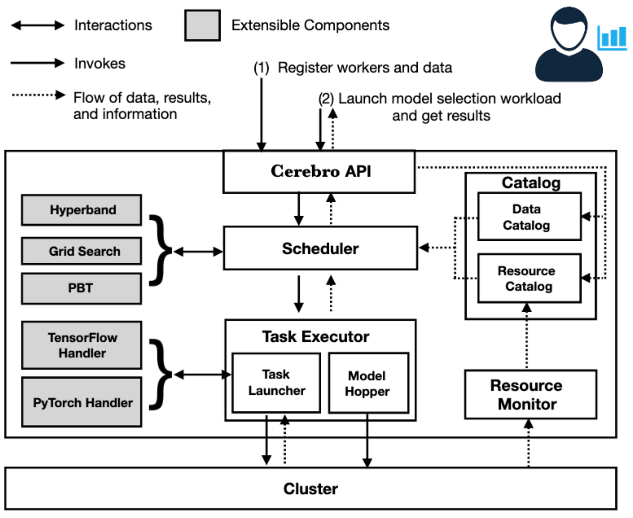

Figure 2: Cerebro System Architecture [4]

Cerebro is a system with five main components: the API, Scheduler, Task Executor, Catalog, and Resource Monitor (see Figure 2). The API is how clients interact with the system and includes useful functions such as executing a training or validation epoch given a set of estimators. The Scheduler is responsible for running randomized scheduling by keeping track of idleness and remaining worker-configuration pairs. The Task Executor is responsible for starting sub-epochs on worker nodes. The Catalog provides worker and data availability information to the Scheduler. The Resource Monitor monitors the cluster of worker nodes through heartbeat checks and updates the Catalog. Adding a new execution backend requires implementing the functionality for all of these components while also conforming to the API.

2.2 Ray

Ray offers a number of primitives that are useful for general-purpose distributed computing, however only a few of these primitives were sufficient for our purposes. The main thing that Ray introduces to users is remote functions, also known as Ray Tasks.

```
@ray.remote
def f(x):
    return x * x

futures = [f.remote(i) for i in range(4)]
print(ray.get(futures)) # [0, 1, 4, 9]
```

In the code above [1], `f` is a remote function that can be called using `f.remote(i)`. The return value of this call is a remote object reference, the significance of which will be explained later in this subsection. The actual return value can be fetched using `ray.get(f.remote(i))`. Calling `f` multiple times simultaneously as a Ray Task rather than an ordinary Python function allows Ray to utilize multiple cores to parallelize execution, thereby cutting down runtime significantly. For example, if the above code were to run `[f(i) for i in range(4)]` instead, it would take about four times longer.

Just as functions can be made remote, classes can as well. In Ray, these are called Actors or stateful workers.

```
@ray.remote
class Counter(object):
    def __init__(self):
        self.value = 0

    def increment(self):
        self.value += 1
        return self.value

# Create an actor from this class.
counter = Counter.remote()

# Call the actor.
obj_ref = counter.increment.remote()
assert ray.get(obj_ref) == 1
```

The code above [2] creates a remote instance of `Counter` using `Counter.remote()`. It has a state, `value`, and a remote function, `increment`, which can be called in the same manner as the remote function seen previously: `counter.increment.remote()`.

As previously mentioned, the return values of Ray Tasks are remote object references. The remote objects themselves are stored in a Pickle format in Ray's shared-memory object stores, of which there is one per node in a Ray cluster. We may not know on which node a remote object lives on; however, it can be replicated in other object stores. The caveat of this is that remote objects are immutable, which is done to remove the need of synchronization after writes. It is not necessary to write remote functions to generate remote objects as users can call `ray.put(obj)` to put objects in Ray's global object store. To delete objects from the object store, the remote object reference simply needs to go out of scope or be deleted using Python's `del` operator.

Ray Data offers a set of functions specifically geared towards datasets. For our purposes, we were only interested in the three following functions.

(1) `read_parquet()`: read a parquet file into memory as an Arrow dataset
(2) `Dataset.split(n)`: split an Arrow dataset into *n* shards
(3) `Dataset.to_pandas()`: convert the Arrow dataset into a Pandas DataFrame

One thing to note is that Arrow tables are columnar, which makes retrieving entire columns quickly to be transformed to tensors for training. The `split()` function is also able to do fast, disjoint sharding across workers.

3 IMPLEMENTATION

In order to add a new execution backend to Cerebro, one must implement a new class that extends the abstract `Backend` class. Specifically, the resulting implemented class should include functionality to both materialize training data (e.g. writing the preprocessed training data to persistent storage such as the Hadoop Distributed File System (HDFS)) and implement MOP. The latter requires running multiple worker services in a cluster and a driver program to act as the Scheduler, Task Executor, Catalog, and Resource Monitor. What follows are the functions implemented in our new `RayBackend` class in the order that they should be called.

3.1 `__init__()`

The constructor of the new execution backend first tries to detect an existing Ray cluster (which is the setting for when Ray has been deployed over multiple machines) and tries to connect to it. If a Ray cluster is not found (specified by a ConnectionError returned by Ray which is handled), then Ray is initialized through the constructor for a single node setting for the machine being executed on. If the number of workers is given, then that number is saved in an instance variable. Otherwise, that number is set to the number of nodes detected. Other instance variables required for executing Model Hopper Parallelism are set to default values.

The constructor also decides the resources to allocate to each worker. This is done for the expectation that in a multi-machine setting, each worker should be assigned the whole machine. Although a rigid requirement, the code allows that this rigidity can later be changed if other flexible approaches are needed later. Using *ray.available_resources()*, the constructor is able to gauge the total number of CPUs available to the ray cluster. It is also able to gauge the total number of machines (nodes) that are deployed in the cluster. With the assumption that the number of workers is the same as the number of machines, the total number of CPUs is equally divided by the number of machines. This *number_of_CPUs_per_worker* is assigned as the number of resources to each worker.

A point to note is that Ray can only provide the total number of CPUs in the cluster, and not the number of CPUs per machine. Thus, this approach works only if the machines the Ray Cluster is deployed on are homogeneous, i.e. they have the same number of CPUs. If machines with different number of CPUs are deployed, the currently used code could have a problem. For example, if there are 2 machines with 10 and 20 CPUs and 2 workers, the constructor will assign 15 CPUs to each worker. The first worker can be deployed using 15/20 CPUs of the machine, but there is no machine that has 15 machines left for the second worker, leading to an infinite wait. If needed, this problem may be mitigated by removing the use of *ray.available_resources()* and instead using third party tools that can provide the number of CPUs per machine. This can be used to dynamically allocate resources to the workers.

In the case of Ray being deployed on a single machine, a much simpler approach is used where the number of CPUs is set as 4. For the simple setting of 4 workers on a single machine, which is the most expected, this was found as the optimal number that reduces latency due to any memory stalls and also does not interfere with other processes on the machines using the other cores.

3.2 `initialize_workers()`

In addition to defining a class for the new Ray execution backend, we also defined a remote class, `Worker`, which has the four following functions.

(1) `__init__()`: Initializes one instance variable which essentially indicates whether this worker is idle.
(2) `get_completion_status()`: Returns whether this worker is idle or not.
(3) `accept_data(data_shard, is_train)`: This function is called when the time comes for the worker to accept its assigned shard of data. `is_train` indicates whether that data is part of the train or validation set.
(4) `execute_subepoch(fn, is_train, initial_epoch)`: Either trains or evaluates on the data already given to this worker and writes the result to storage.

This function creates a list of these remote workers, with the size being the one specified in the constructor. The instance variable specifying whether the workers have been initialized is set to `True`.

3.3 `prepare_data()`

Given a Pandas DataFrame and storage object, this function first writes the DataFrame to the store in Parquet format, splits the DataFrame into train and test sets, and creates a Python dictionary that is populated with the metadata of each column. This ensures recovery from possible node failures.

3.4 `get_metadata_from_parquet()`

This function reads the Parquet files written to the store from the previous function, and generates and returns the same metadata dictionary. The metadata includes the type and size of each column in the dataset.

3.5 `initialize_data_loaders()`

Only if the workers are initialized (determined by checking an instance variable), this function starts by reading the Parquet files written to the store by `prepare_data()` using the function to read Parquet files in Ray Data. The result is an Arrow dataset, which can be split into a number of shards equal to the number of workers. Each worker then receives one shard via the `accept_data` function. Finally, an instance variable indicating that the data loaders are initialized is set to `True`.

3.6 `train_for_one_epoch()`

This is the main function for carrying out the Model Hopper Parallelism over all models 5, checkpointing the models trained for each epoch, and returning the epoch results such as loss and accuracy back to the caller.

What follows is our explanation of some primitives and functions we use to implement this training function.

(1) Ray Estimator: In the original Spark implementation, the Keras models as well as other objects that were used to train it (optimizer, criterion function, store object, etc.) were wrapped in `SparkEstimator` instances which were then passed into the *train_for_one_epoch()* function (In the subsequent subsections we will show how this estimator is used).

To remove this Spark dependency, we had to create our own `RayEstimator` class. This Ray Estimator includes the model, all the keras objects required for training (optimizer, loss, etc.) as well as functions for compiling the model, getting Keras Utilities to load and save the model, etc. Other functions that are added in this class included getters and setters for all the objects. Essentially, this is a wrapper that allows us to treat the whole model (with its losses, optimizers etc.) as an atomic unit.

(2) `get_remote_trainer()`: The *train_for_one_epoch()* function begins by getting some metadata from the models and data and uses this to create training instances of each model provided to the function. The creation of training instances is done through the *get_remote_trainer()* function, which checkpoints the model from the Ray Estimator (explained below) object if it has not already been check-pointed. It then loads the model, compiles it and returns it in a training instance, called a *sub_epoch_trainer*.

(3) `sub_epoch_trainer()`: The *sub_epoch_trainer* loads all the primitives needed for trainning the model and keeps them as a part of the function. These include the callbacks during Keras training specified by the user, the functions for training and evaluating the model from Keras, the remote store that contains all the paths to the models, and the Keras model deserialization functions. It also embeds the training function used for training the sub-epoch. This training function can be called by the Ray Workers, and loads the model from the object stores (as described below) trains them on the sub-epoch data and returns the sub-epoch result. Each subepoch trainer contains this data for its respective model and returns the train function that can be executed by the worker.

train_for_one_epoch uses the functions above and schedules them using a version of the randomized scheduling algorithm (see Figure 3). The function first calls *get_remote_trainer()* and for the estimators returned by the model, each model is put into the Ray Object Store using *ray.put()*. This gives the Object References for each model to the master function, which can pass on these object references to the worker functions. The worker functions can obtain the underlying model using *ray.get()*. However, during the implementation, it was not so simple. We encounter a huge problem that while the *get_remote_trainer()* returns compiled models to us, these Keras models are not picklable. On the other hand, Ray uses pickling to place the objects in the object stores. We, thus, cannot place the compiled models into the object store. Instead, we have to break down the model into its architecture (stored as a JSON file using *model.to_json()*, the model's constituent weights (obtained using textitmodel.get_weights()), the loss and optimizer functions, and the optimizer states. These are all stored in a dictionary and sent to the object store. When any process tries to get the model, it calls *ray.get()* on the object reference and obtains this dictionary. It then gets the respective architecture, weights, loss and optimizers, and uses Keras's compile function to get the model it can train. As expected, this whole process incurs a huge overhead that can be solved if `tf.keras` models become picklable in the future. Finally, the *train_for_one_epoch* function is able to pass all the initial model references to the object store and begins scheduling the models on the workers according to the randomized scheduling.

Algorithm 1 Randomized Scheduling

```
Input: $S$
$Q = \{s_{i,j} : \forall i \in [1, \ldots, |S|], \forall j \in [1, \ldots, p]\}$
worker_idle $\leftarrow$ [true, ..., true]
model_idle $\leftarrow$ [true, ..., true]
while not empty($Q$) do
  for $j \in [1, \ldots, p]$ do
    if worker_idle[$j$] then
      $Q \leftarrow$ shuffle($Q$)
      for $s_{i,j'} \in Q$ do
        if model_idle[$i$] and $j' = j$ then
          Execute $s_{i,j'}$ on worker $j$
          model_idle[$i$] $\leftarrow$ false
          worker_idle[$j$] $\leftarrow$ false
          remove($Q, s_{i,j'}$)
          break
  wait WAIT_TIME
```

Algorithm 2 When $s_{i,j}$ finishes on worker j

```
model_idle[$i$] $\leftarrow$ true
worker_idle[$j$] $\leftarrow$ true
```

Figure 3: Randomized Scheduling Algorithm [4]

We can merge the two algorithms shown in Figure 3 to get our version of the randomized scheduling. What was Algorithm 2 would be executed if the jth worker is not idle according to the local array, but has completed a subepoch, which can be checked using `get_completion_status()`. In addition to changing the local `*_idle` arrays, the randomized scheduling algorithm that we implemented would try to find an idle worker that the newly idle model has not yet visited and send it to that worker if it exists. Essentially, this is the heartbeat check used by the Resource Monitor to check for failures, but instead we use it to check completion statuses of workers. Due to time constraints, we were not able to check for and recover from failures, which we will discuss more about in Section 5.

Figures 4 and 5 show the training process on the worker when a model is assigned on a worker. The worker gets the reference to the model which is included in the *sub_epoch_trainer* as well as other useful data such as if it is for training or validation and the epoch number. It then gets the training/validation data shard assigned to it (present in its state), and sends all of this to the train function of the *sub_epoch_trainer*. The train function is able to get the model by sending the model reference to the object store. Of course as described before, the function gets the dictionary whose elements it has to pass to the Keras compile function to get the model. Once it gets this model, it trains it using Keras' fit function on the sub-epoch data. The worker now has the trained model and it sends this back to the object store (splitting it up into a dictionary as described before). The previous model (that the worker initially got) now has no references pointing to it and is garbage collected

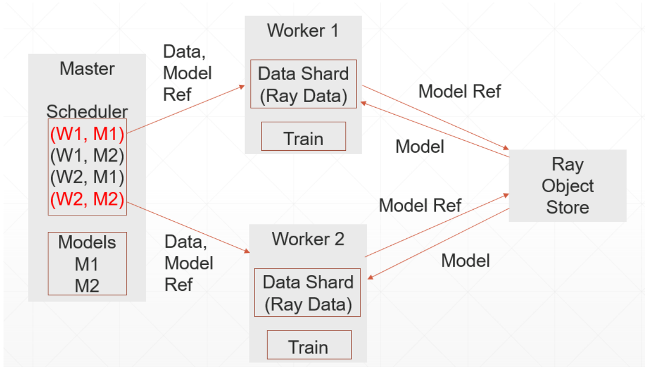

Figure 4: Training Process on the Worker When Getting Model

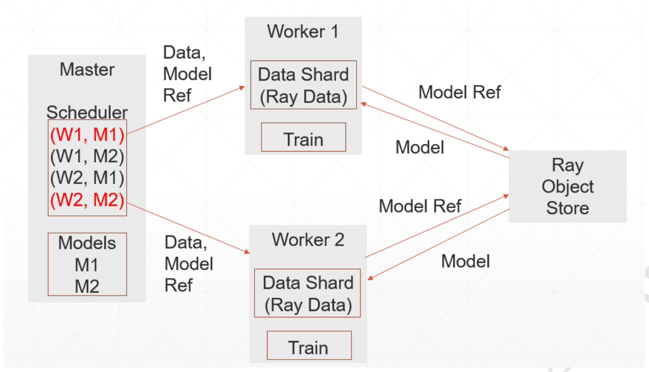

Figure 5: Training Process on the Worker When Subepoch Training Completes

by Ray. This ensures that the Ray Object store has only one copy of a model at a time and does not run out of memory. The worker then sends the epoch results (loss, accuracy, etc) along with the latest model's Object reference back to the *train_for_one_epoch()* function which can use these model references to assign the model to the next worker.

Once the entire scheduling is finished, the *train_for_one_epoch()* function gathers all the sub-epoch statistics for each model and averages them to get the epoch statistics. It also increments each model's epoch by 1. It then sends the epoch statistics to the callee tune function which uses them for logging metrics.

However, we realised that using the Ray Object store, where we have to split up the model into a dictionary and compile it every time we receive it, incurs a huge overhead. In the final days, we tried using just the remote object store provided in `cerebro.storage`, as is done in the Spark implementation. Here we checkpoint the model using *model.save()* into a path in the object store and load it from this path again using the *keras_deserialize_fn()* implemented in the backend, just as it is done in the Spark implementation. We find that this is actually faster than the Object Store implementation, and shaves around 1 second off per epoch while training. Hence, this is the approach we use in the final code that we have submitted.

3.7 `teardown_workers()`

This function only calls `ray.shutdown()` in order to shut down all Ray processes in the cluster. This automatically kills all the consistent worker processes deployed on different machines and shuts down the master Ray process, before returning control to the calling `tune.py` function.

4 RESULTS

We try to evaluate our model on 2 datasets: An augmented version of the MNIST dataset and the Criteo dataset. While we get promising results on the augmented MNIST as well as a truncated Criteo dataset, we run into hardware issues (which we have elaborated below) while evaluating on the full Criteo dataset. Hence, we are unable to show the final results of replication of the Criteo in [4]. However we will explain why we believe we can replicate those results once the hardware issues are resolved.

4.1 Augmented MNIST

The MNIST dataset consists of 24x24 sized images that consist of handwritten digits, while the labels are 0-9. It is a classic classification task used to test neural networks. The original MNIST dataset is considerably tiny for our use case, containing only 70000 data points (feature images and labels). We instead use an augmented version of the dataset that increases the data by 10 times to 700000 data points. This is more suited to the scale at which we are doing distributed training. We flatten the input images into vectors with 784 dimensions and use a fully connected neural network with 2 layers (hidden sizes of 1000 and 500) for classification into the 10 labels. For the loss function we use Keras' `CategoricalCrossentropy` loss (as the labels are one-hot vectors) and the optimizer used is the Adam optimizer.

We use 4 workers for our implementation (each worker is assigned one full machine). We use 4 different hyperparameter configurations for this testing, spanning across 2 learning rates and 2 regularization values. In total, we train 4 models on Criteo. We compare our approach with 2 other approaches. The first is a naive sequential approach that uses Keras to train each model one after the other. The second comparison is to Ray Tune [3], which is a task parallel approach that replicates the data across multiple workers, assigns a model to each worker and trains it end-to-end.

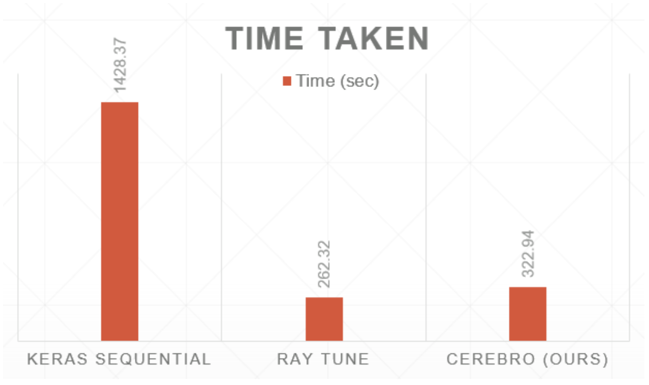

Figure 6: MNIST Time Bar Chart

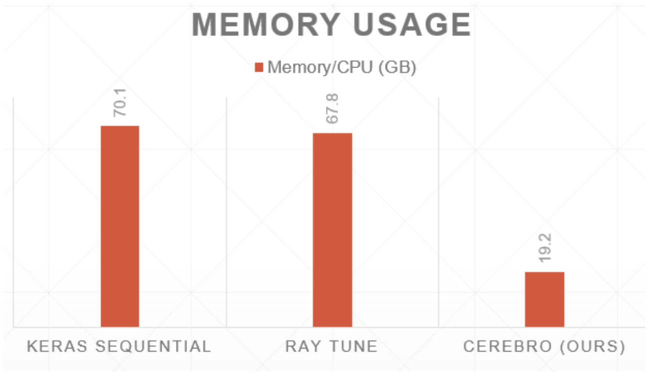

Figure 7: MNIST Memory Bar Chart

The Figures 6 and 7 show the results of our implementation's performance with the other approaches. Looking at the time taken for training the 4 models (Figure 6), we take 322.94 seconds compared to the sequential model's 1428.37 seconds, which is more than a 4x speedup. Considering that our network uses 4 models, we are thus able to provide more than a linear speedup compared to sequential models. However, we perform slightly worse than Ray Tune, which takes 262.37 seconds. The reason for this is that the data we use is able to fit on a single node of the machine, essentially making this problem task parallel. In this case, we incur the overhead of Model Hopping, while Tune is able to train the models end-to-end and thus perform better. We believe that combining this with a data parallel approach, where the data does not fit on one node, will lead Tune to incur I/O overhead of loading the data shards again and again. This is much larger than the MOP overhead and will lead to our model performing again.

Figure 7 shows the memory usage of the 3 different approaches. We are able to provide almost a 4x decrease in the resource usage per CPU. This is because the other approaches load the entire data into memory of a machine, while we shard the data across the workers. Since we use 4 workers, we store one-fourth of the data on each worker, thus providing an efficient memory usage.

Figures 8 and 9 are sanity checks to show that all the 3 approaches follow sequential SGD. As we can see, the loss curves of all 3 approaches are quite similar and converge quickly, while all the approaches approach 98-99% validation accuracy within 5 epochs.

4.2 Criteo

For Criteo, we run into a hardware problem while evaluating on the dataset. Each partition of the train data is too large to fit in memory, and thus when we want to transform it into the parquet files for Cerebro (we need to make changes like changing the 'labels' column to 'label', making the one-hot-encoded list of labels into Numpy arrays and putting this transformed data back to a parquet file), we ran out of RAM and the program crashed. This was despite trying to transform the data with multiple libraries like Pandas, Ray Data, etc. We tried with machines up to 156 GB of RAM and got out-of-memory issues. Since this testing was in the last 2 weeks of the quarter, we were unable to get access to machines with more RAM.

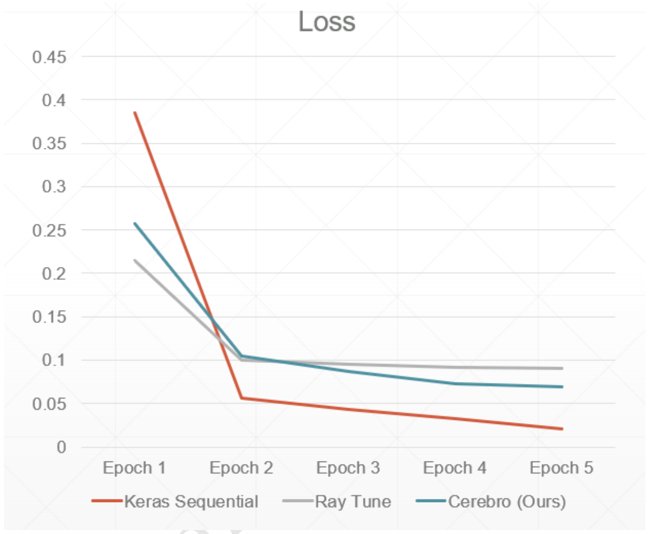

Figure 8: MNIST Loss Plot

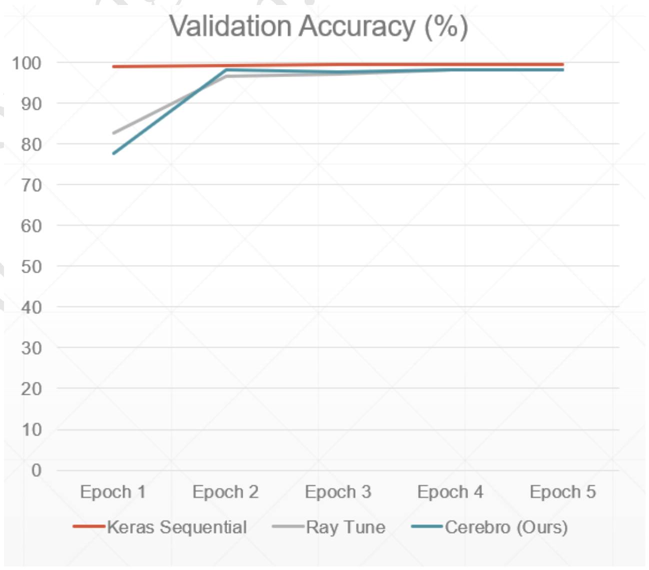

Figure 9: MNIST Accuracy Plot

However, we did try to use a very small subsample of the Criteo dataset (by getting access to the original TSV files, removing all the categorical values and using a 10% sample of a single training partition). Using this, we were able to get increases in performance (memory and time) comparable to our MNIST example. However, all 3 approaches were not able to converge or get any reasonable accuracy as we were training on only the numeric features while discarding categorical ones. Looking at the performance benefits, we believe that if suitable hardware is accessed, we should be able to replicate the results of [4] on our Ray implementation.

In the current code we submit, we have only shown the MNIST example, as we feel the Criteo example is incomplete if we don't have comprehensive results. However, provided that the Criteo dataset is suitably prepared for our implementation, an evaluation script for it can be easily generated by following our MNIST example.

5 FUTURE WORK

We plan to implement replica-aware scheduling as mentioned in [4]. This would involve replicating shards across multiple workers in the hopes that configurations would not need to visit all workers at each epoch. This would require the remote `Worker` instances to maintain a list, set, or dictionary of shards instead of having just two references to its train and validation data shards. The Scheduler would also need as input an availability mapping of shards to workers, or the workers can expose what shards are in their possession through a function which the Scheduler can call at the beginning of every epoch to construct the mapping itself. Whenever the Scheduler looks for a worker which a configuration has not visited, it will simply consult the availability map.

We plan to implement fault tolerance and elasticity, also mentioned in [4] which suggests keeping track of subepochs currently being trained. If a failure is discovered during a heartbeat check, the subepoch will be moved back into the original set of subepochs to be run. The Scheduler can find another node with a replica of the shard and use the last checkpoint of the configuration written to the store to proceed.

Future work also includes the goals of Cerebro that can be implemented on Ray. The first is hybridizing MOP with model parallelism, which is sharding the models themselves across nodes. The second is including support for more complex model selection scenarios such as transfer learning.

6 CONCLUSION

Ultimately, the Ray API greatly simplified the task of implementing MOP. Since Ray abstracted out most of the low level implementation details such as sending data over a wire, serialization, and synchronization we were largely able to focus on the high level components of the Cerebro system. We were also able to see firsthand the effectiveness of MOP compared to those of sequential training and task parallelism and identify pressure points of dealing with datasets that don't fit within a single node's memory. In the future, we aim to resolve the issues that we ran into and introduce the mentioned optimizations to improve throughput of model selection using Ray.

ACKNOWLEDGMENTS

We are grateful to Professor Arun Kumar for meeting with us weekly and guiding us through the entire process, and to Supun Nakandala and Yuhao Zhang for answering our technical questions.

REFERENCES

[1] [n. d.]. A Gentle Introduction to Ray. https://docs.ray.io/en/latest/ray-overview/index.html.

[2] [n. d.]. Ray Core Walkthrough. https://docs.ray.io/en/latest/walkthrough.html.

[3] Richard Liaw, Eric Liang, Robert Nishihara, Philipp Moritz, Joseph E Gonzalez, and Ion Stoica. 2018. Tune: A research platform for distributed model selection and training. *arXiv preprint arXiv:1807.05118* (2018).

[4] Supun Nakandala, Yuhao Zhang, and Arun Kumar. 2020. Cerebro: A Data System for Optimized Deep Learning Model Selection.