Apache Airflow and Ray
Orchestrating ML at Scale
Background

• Strategy Engineer at astronomer.io, Airflow PMC member, co-creator of K8sExecutor

• Previously: Building data science platforms at Bloomberg LP

• Obsessed with Data Science Tooling, and building distributed systems
The Airflow Data Science Story
The Airflow Data Science Story

- Airflow is the tool to take you from experiment to production model
- Monitoring and scheduling ensure your models update in time for SLAs
- Connection handling for easily switching between dev -> prod data sources
- Fault tolerant scheduler that can retry jobs in case of failure
The Airflow Data Science Story
The (Traditional) Airflow Data Science Story

Experiment

Parameterize

Productionize
• Parameterize Notebooks through cell tagging
• Stores intermediate notebooks
• Execute using Python API or CLI
• Stores notebooks to S3/GCS
Issues with this approach

- Entire notebook executes as a single task
- Low visibility, no fault tolerance
- Code is in multiple locations
- Experimentation becomes difficult
- Repeatability becomes messy
The Next Gen Airflow Data Science Story?

<table>
<thead>
<tr>
<th>Experiment</th>
<th>Parameterize</th>
<th>Productionize</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Jupyter logo" /></td>
<td>( __ (ツ)__ _ _</td>
<td>( __ (ツ)__ _ _</td>
</tr>
</tbody>
</table>
The Ideal Story

• Minimal conversion from Jupyter notebook -> Airflow DAG
• Moving large datasets between different tasks should be trivial.
• Should be able to request dedicated resources for the compute job
  • GPU, RAM, CPU, etc.
• Register & deploy, and replicate the resulting models.
• Maintain orchestration and monitoring at scale.
Enter Ray

“[A] distributed execution framework that makes it easy to scale your applications and to leverage state of the art machine learning libraries."
Enter Ray

- Run the same code on your local machine, on an EC2 VM, a hardware machine, etc. with no code change!
- Native Integrations with many ML projects
- Simple setup and native pythonic library
- Options for distributed computation (dask, spark, modin)
- Ray Serve for model serving
The Ideal Story
The Taskflow API

- Introduced Airflow 2.0
- Convert a python function to an Airflow task using just a single decorator!
- Pass data between tasks using functional composition
- But what if we could add the power of Ray?

```python
@task
def get_initial_number():
    return 1

@task
def add_one(value):
    return value + 1

@dag(dag_kwargs=dag_kwargs)
def dag():
    value = get_initial_number()
    for i in range(5):
        value = add_one(value)
```
The Ideal Story

@task

@ray.remote(num_cpu=2)
Introducing The Ray Decorator!

@ray_task
def get_initial_number():
    return 1

@ray_task(num_cpu=2)
def add_one(value):
    return value + 1

@dag(dag_kwargs=dag_kwargs)
def dag():
    value = get_initial_number()
    for i in range(5):
        value = add_one(value)
Introducing The Ray Decorator!

- Automatically run your airflow tasks in your ray cluster with one line of code!
- Ability to dynamically size tasks and access large ray instances
- Intermediate values automatically stored in the plasma store for ease and data locality!

```python
@ray_task
def get_initial_number():
    return 1

@ray_task(num_cpu=2)
def add_one(value):
    return value + 1

@dag(dag_kwargs=dag_kwargs)
def dag():
    value = get_initial_number()
    for i in range(5):
        value = add_one(value)
```
The top-tier ML tooling of Ray with the Stability and Ecosystem of Airflow
From Notebook to Production
Develop

```
@ray.remote
def do_some_stuff():
    ...

do_some_stuff.remote()

@ray.remote
def do_some_stuff():
    ...

return data
do_some_stuff.remote()
```
@ray_task
def train_model():
    ...

@dag(...)
def my_dag():
    train_model()
Parameterize

data_path = }
“{{ conf.data_path }}”

@ray_task
def train_model(path: str):
   ...

@dag(...)
def my_dag():
   train_model(data_path)
Productionize

Deploy your DAG!

Now it’s easy to:
- Add more tasks & parallelize
- Tune the model(s)
- Schedule fresh updates
- Monitor for failures
- (Re)Deploy the best model(s)
- Connect to the ecosystem
The Next Gen Airflow Data Science Story

Develop

Experiment & Parameterize

Productionize

Native Libraries
- tune
- rllib
- rayserve
- raysgd

3rd Party Libraries
- spaCy
- dmlc
- XGBoost
- Scikit
- ClassyVision
- Optuna
- MARS
- W&B

Ray
Framework for distributed Python

AWS
Google Cloud
Microsoft Azure
Databricks
Confluent
How It Works
How It Works
How It Works

Start task 1

Alert Airflow of function completion and send object ID
Send ray.remote function

Store result

Store Object ID
How It Works

Task 2

Send ray.remote function with object ID

Retrieve Object using ID

Retrieve Object ID
How It Works
Next Steps
Checkpointing

- Store intermediate data in external data stores
- Re-run failed tasks
- Plug Tune checkpoints to model registries and experiment tracking libs
- Tweak Experiments so even if your ray cluster crashes, you will be able to restart DAG from checkpoint

```python
@ray_task(checkpoint=True)
def really_long_model():
    ...

@ray_server_task
def serve_model():
    ...

@dag(dag_kwargs=dag_kwargs)
def dag():
    model = really_long_model()
    serve_model(model)
```
Ray serve decorator

• Deploy models to your ray cluster via airflow DAGs for instance prediction endpoints

• Composed Models = Multiple models based on business logic

• Parallelize multi-model training with Airflow

```python
@ray_task
def create_model_1():
    ...
@ray_task
def create_model_2():
    ...

@ray_serve_task
def serve_model():
    ComposedModel.deploy()

@dag(dag_kwargs=dag_kwargs)
def dag():
    model = create_model()
    serve_model(model)
```
Road Map

- May 2021
  - Alpha released
  - Modin Support
  - Custom sizes

- July 2021
  - Beta released
  - Checkpointing
  - Anyscale connector

- Q3 2021
  - GA released
  - Fully tested and supported

- Q4 2021
  - GA Running with Anyscale cloud
How to Get the Ray Provider
How to Get the Ray Provider

Head to https://registry.astronomer.io/

Providers
Python packages containing all relevant Airflow modules for a third-party service.
How to Get the Ray Provider

`pip install airflow-provider-ray`
Thank You

@danimberman
@ApacheAirflow

astronomer.io
@astronomerio

Special thanks to:
• Richard Liaw
• Will Drevo
• Charles Greer
• Pete DeJoy
• Rob Deeb
• Plinio Guzman