Cloudera Data Science Workbench

Distributed Computing with Workers

Date published: 2020-02-28
Date modified:

https://docs.cloudera.com/

Legal Notice

Contents

Distributed Computing with Workers

Cloudera Data Science Workbench provides basic support for launching multiple engine instances, known as workers, from a single interactive session. Any R or Python session can be used to spawn workers. These workers can be configured to run a script (e.g. a Python file) or a command when they start up.

Workers can be launched using the launch_workers function. Other supported functions are list_workers and stop _workers. Output from all the workers is displayed in the workbench console of the session that launched them. These workers are terminated when the session exits.

Using Workers for Machine Learning

The simplest example of using this feature would involve launching multiple workers from a session, where each one prints 'hello world' and then terminates right after. To extend this example, you can remove the print command and configure the workers to run a more elaborate script instead. For example, you can set up a queue of parameters (inputs to a function) in your main interactive session, and then configure the workers to run a script that pulls parameters off the queue, applies a function, and keeps doing this until the parameter queue is empty. This generic idea can be applied to multiple real-world use-cases. For example, if the queue is a list of URLs and the workers apply a function that scrapes a URL and saves it to a database, CDSW can easily be used to do parallelized web crawling.

Hyperparameter optimization is a common task in machine learning, and workers can use the same parameter queue pattern described above to perform this task. In this case, the parameter queue would be a list of possible values of the hyperparameters of a machine learning model. Each worker would apply a function that trains a machine learning model. The workers run until the queue is empty, and save snapshots of the model and its performance.

Workers API

This section lists the functions available as part of the workers API.

Launch Workers

Launches worker engines into the cluster.

Syntax

```
launch_workers(n, cpu, memory, nvidia_gpu=0, kernel="python3", s
cript="", code="", env={})
```

Parameters

- n (int) - The number of engines to launch.
- cpu (float) - The number of CPU cores to allocate to the engine.
- memory (float) - The number of gigabytes of memory to allocate to the engine.
- nvidia_gpu (int, optional) - The number of GPU's to allocate to the engine.
- kernel (str, optional) - The kernel. Can be "r", "python2", "python3" or "scala". This parameter is only available for projects that use legacy engines.
- script (str, optional) - The name of a Python source file the worker should run as soon as it starts up.
- code (str, optional) - Python code the engine should run as soon as it starts up. If a script is specified, code will be ignored.
- env (dict, optional) - Environment variables to set in the engine.

Example Usage

Python

```
import cdsw
workers = cdsw.launch_workers(n=2, cpu=0.2, memory=0.5, code="pr
int('Hello from a CDSW Worker')")
```

R

```
library("cdsw")
workers <- launch.workers(n=2, cpu=0.2, memory=0.5, env="", cod
e="print('Hello from a CDSW Worker')")
```

Note: Due to a bug, the env parameter must been defined when calling the launch.workers function in R. If you do not wish to pass environment variables, simply set it to an empty string. When not defined, the env parameter is serialized internally into a format that is incompatible with Cloudera Data Science Workbench. This bug does not affect the Python engine.

List Workers

Returns all information on all the workers in the cluster.

Syntax

```
list_workers()
```

Await Workers

Waits for workers to either reach the running status, or to complete and exit.

Syntax

```
await_workers(ids, wait_for_completion=True, timeout_seconds=60)
```

Parameters

- ids: int or list of worker descriptions, optional The id's of the worker engines to stop or the worker's description dicts as returned by launch_workers or list_workers. If not provided, all workers in the cluster will be stopped.
- wait_for_completion: boolean, optional If True, will wait for all workers to exit successfully. If False, will wait for all workers to reach the running status. Defaults to True.
- timeout_seconds: int, optional Maximum number of seconds to wait for workers to reach the desired status. Defaults to 60. If equal to 0, there is no timeout. Workers that have not reached the desired status by the timeout will be returned in the failures key. See the return value documentation.

Returns

- dict - A dict with keys workers and failures. The workers key contains a list of dicts describing the workers that reached the desired status. The failures key contains a list of descriptions of the workers that did not.
-  **Note:** If wait_for_completion is False, the workers in the 'workers' key will contain a key called 'ip_address' which contains each worker's external IP address. This can be useful for running distributed frameworks on workers.

Stop Workers

Stops worker engines.

Syntax

```
stop_workers(*worker_id)
```

Parameter

- worker_id (int, optional) - The ID numbers of the worker engines that must be stopped. If an ID is not provided, all the worker engines on the cluster will be stopped.

Example: Worker Network Communications

Workers are a low-level feature to help use higher level libraries that can operate across multiple hosts. As such, you will generally want to use workers only to launch the backends for these libraries.

To help you get your workers or distributed computing framework components talking to one another, every worker engine run includes an environmental variable CDSW_MASTER_IP with the fully addressable IP of the master engine. Every engine has a dedicated IP access with no possibility of port conflicts.

For instance, the following are trivial examples of two worker engines talking to the master engine.

R

From the master engine, the following master.r script will launch two workers and accept incoming connections from them.

```
# master.r

library("cdsw")
# Launch two CDSW workers. These are engines that will run in
# the same project, execute a given code or script, and exit.
workers <- launch.workers(n=2, cpu=0.2, memory=0.5, env="", script="worker
.r")

# Accept two connections, one from each worker. Workers will
# execute worker.r.
for(i in c(1,2)) {
  # Receive a message from each worker and return a response.
  con <- socketConnection(host="0.0.0.0", port = 6000, blocking=TRUE, server
=TRUE, open="r+")
  data <- readLines(con, 1)
  print(paste("Server received:", data))
  writeLines("Hello from master!", con)
  close(con)
}
```

The workers will run the following worker.r script and respond to the master.

```
# worker.r

print(Sys.getenv("CDSW_MASTER_IP"))
con <- socketConnection(host=Sys.getenv("CDSW_MASTER_IP"), port = 6000, bloc
king=TRUE, server=FALSE, open="r+")
write_resp <- writeLines("Hello from Worker", con)
server_resp <- readLines(con, 1)
print(paste("Worker received:  ", server_resp))
close(con)
```

Python

From the master engine, the following master.py script will launch two workers and accept incoming connections from them.

```
# master.py
import cdsw, socket
```

```
# Launch two CDSW workers. These are engines that will run in
# the same project, execute a given code or script, and exit.
workers = cdsw.launch_workers(n=2, cpu=0.2, memory=0.5, script="worker.py")

# Listen on TCP port 6000
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind(("0.0.0.0", 6000))
s.listen(1)

# Accept two connections, one from each worker. Workers will
# execute worker.py.
conn, addr = s.accept()
for i in range(2):
    # Receive a message from each worker and return a response.
    data = conn.recv(20)
    if not data: break
    print("Master received:", data)
    conn.send("Hello From Server!".encode())
conn.close()
```

The workers will run the following worker.py script and respond to the master.

```
# worker.py
import os, socket

# Open a TCP connection to the master.
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((os.environ["CDSW_MASTER_IP"], 6000))

# Send some data and receive a response.
s.send("Hello From Worker!".encode())
data = s.recv(1024)
s.close()

print("Worker received:", data)
```