The AI Thunderdome

Using OpenStack to accelerate AI training with Sahara, Spark, and Swift

Sean Pryor, Sr. Cloud Consultant, RHCE
Red Hat
https://www.redhat.com
spryor@redhat.com

Overview

This talk will cover

- Brief explanations of ML, Spark, and Sahara
- Some notes on preparation for Sahara
- (And some issues we hit in our lab while preparing for this talk)
- A look at Machine Learning concepts inside Spark
- Cross Validation and Model Selection
- Sparkflow architecture
- Example code

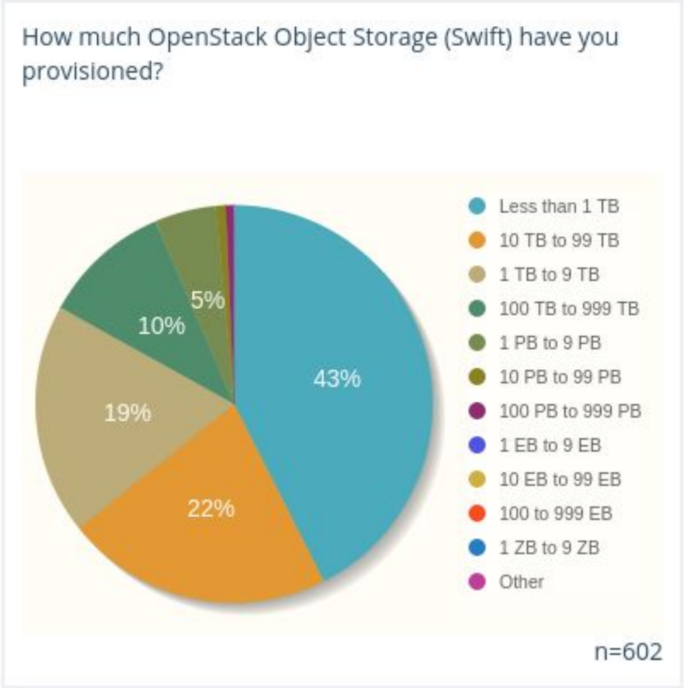
Big Data and OpenStack

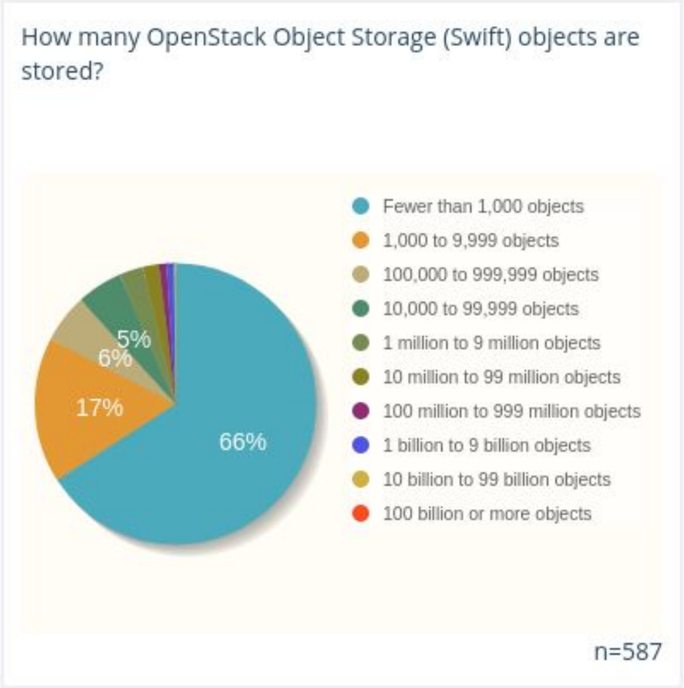
Big Data and OpenStack

From the user survey:
https://www.openstack.org/analytics

A lot of data resides on OpenStack already

- The data is already there. Why move it elsewhere to analyze it?
- Tools are already there to do the analysis

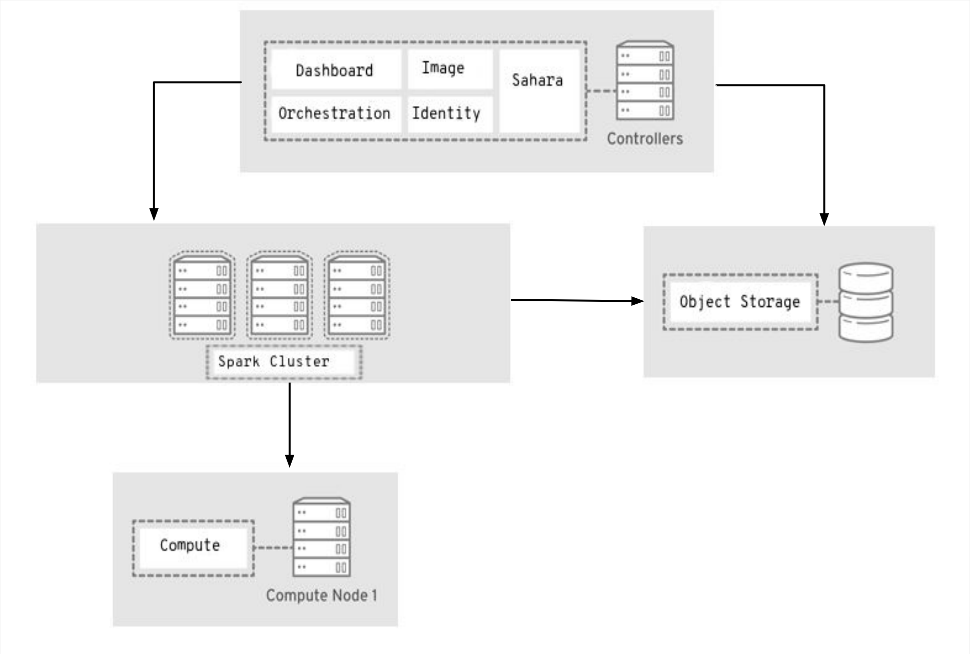




Sahara+Spark+Swift Architecture

Basic architecture outline

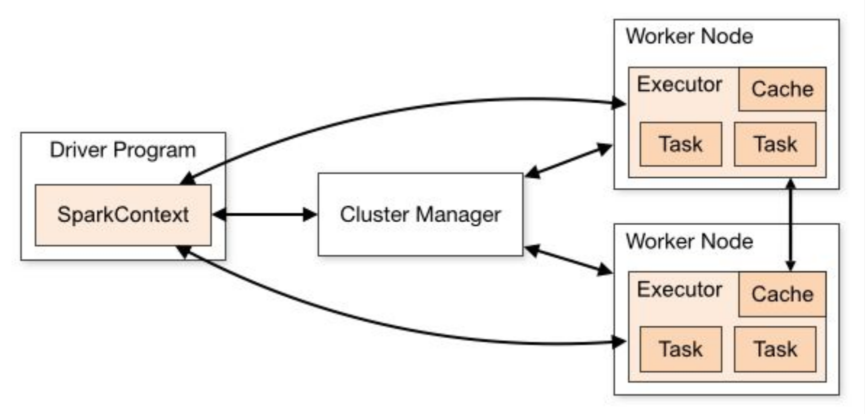
- Sahara is a wrapper around Heat
  - It does more than just Spark too
- Basic architecture involves just Spark on compute nodes
- Spark cluster can directly access Swift via swift://container/object URLs
- Code deployed on Spark clusters can access things independently as well



Spark Architecture Overview

Basic architecture outline

- Spark has a master/slave architecture
- The cluster manager can be either the built-in one, Mesos, Yarn, or Kubernetes
- Spark is built on top of the traditional Map/Reduce framework, but has additional tools, notably ones that include Machine Learning
- For TensorFlow, there are several frameworks that make training and deploying models on Spark a lot easier
- Workers have in-memory data cache - this is important to know when using TensorFlow



Deploying Sahara

A few notes when deploying Spark clusters via Sahara

Image modifications are needed

- guestmount works great here
- pip install:
  - `tensorflow or tensorflow-gpu`
  - `keras`
  - `sparkdl`
  - `sparkflow`
- Add supergroup to ubuntu user

Ensure hadoop swift support is present

- `java.lang.RuntimeException: java.lang.ClassNotFoundException: Class org.apache.hadoop.fs.swift.snative.SwiftNativeFileSystem not found`
- This error indicates support is missing, may need to reinstall /usr/lib/hadoop-mapreduce/hadoop-openstack.jar

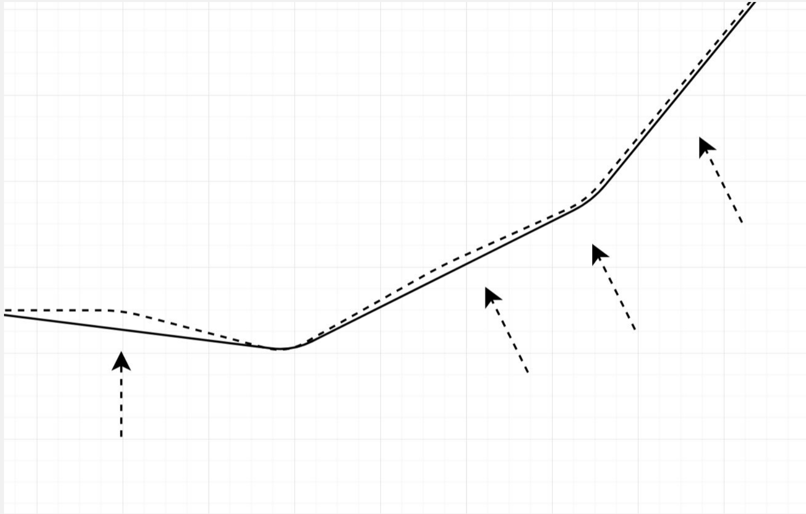
OpenStack job framework doesn't support Python

- The Job/Job Execution/Job Template framework assumes java
- In order to do python, it likely means spark-submit

Machine Learning with Spark

Training AI

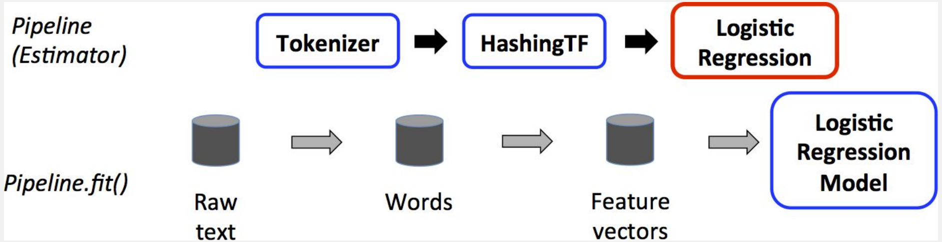
Basic overview of AI and AI training



- For ML techniques, broadly, each iteration tries to fit a function to the data.
- Each new iteration refines the function
- **Features:** Characteristics of a single datapoint
- **Labels:** Outputs of a Machine Learning model
- **Learning rate:** How much each new iteration changes the function
- **Loss:** How far from reality each label is
- **Normalization:** Penalizes complex functions. This helps prevent overfitting

Spark Machine Learning

Important Components in Spark ML



DataFrame

- Built on the regular Spark RDD/DataFrame API
- SQL-like
- Lazy evaluation
- Notably transform() doesn't trigger evaluation. Things like count() do
- Supports a Vector type in addition to regular datatypes

Transformer

- Transformers add/change data in a dataframe
- Transformers implement a transform() method which returns a modified DataFrame

Estimator

- Estimators are Transformers that instead output a model
- Estimators implement a fit() method which trains the algorithm on the data
- Estimators can also give you data about the model like weights and hyperparameters
- Can be saved/reused

Cross Validation

Automatic selection of the best model

- CrossValidator allows you to select model parameters based on results of parallel training
- Wraps a Pipeline, and executes several pipelines in parallel with different parameters
- Requires a grid of parameters to train against
- Splits the dataset into N folds, with a ⅔ train ⅓ test split
- Requires a loss metric to optimize against, Evaluator classes have these pre-baked
- After evaluating on all sets of parameters, the best is trained and tested against the entire dataset
- Parameter grid should ideally be small
- The folding of the dataset means that it's not ideal for small datasets
- Still requires some expertise in making sure it doesn't overfit, or that other errors don't occur

Example Code

Right out of the manual:
https://spark.apache.org/docs/2.3.0/ml-tuning.html

Parallel Hyperparameter Training

Spark CrossValidation Sample Code

```
from pyspark.ml import Pipeline
from pyspark.ml.classification import LogisticRegression
from pyspark.ml.evaluation import BinaryClassificationEvaluator
from pyspark.ml.feature import HashingTF, Tokenizer
from pyspark.ml.tuning import CrossValidator, ParamGridBuilder

training = spark.createDataFrame([
    (0, "a b c d e spark", 1.0),
    (1, "b d", 0.0),
    ...
], ["id", "text", "label"])

tokenizer = Tokenizer(inputCol="text", outputCol="words")
hashingTF = HashingTF(inputCol=tokenizer.getOutputCol(),
outputCol="features")
lr = LogisticRegression(maxIter=10)
pipeline = Pipeline(stages=[tokenizer, hashingTF, lr])

paramGrid = ParamGridBuilder() \
    .addGrid(hashingTF.numFeatures, [10, 100, 1000]) \
    .addGrid(lr.regParam, [0.1, 0.01]) \
    .build()
```

```
crossval = CrossValidator(
           estimator=pipeline,
           estimatorParamMaps=paramGrid,
           evaluator=BinaryClassificationEvaluator(),
           numFolds=2)  # use 3+ folds in practice
cvModel = crossval.fit(training)

test = spark.createDataFrame([
    (4, "spark i j k"),
    (5, "l m n"),
    (6, "mapreduce spark"),
    (7, "apache hadoop")
], ["id", "text"])

prediction = cvModel.transform(test)
selected = prediction.select("id", "text", "probability",
"prediction")
for row in selected.collect():
    print(row)
```

Parallel Hyperparameter Training

Spark CrossValidation Sample Code

- Boilerplate start sets up Spark Session and training data
- Tokenizer takes in the input strings and outputs tokens
- HashingTF generates features by hashing based on the frequency of the input
- LogisticRegression is one of the pre-canned ML algorithms
- Pipeline sets up all the stages

```
from pyspark.sql import SparkSession
from pyspark.ml import Pipeline
from pyspark.ml.feature import HashingTF, Tokenizer
spark = SparkSession.builder.appName("SparkCV").getOrCreate()

training = spark.createDataFrame([
    (0, "a b c d e spark", 1.0),
    (1, "b d", 0.0),
    ...
], ["id", "text", "label"])

tokenizer = Tokenizer(inputCol="text", outputCol="words")


hashingTF = HashingTF(inputCol=tokenizer.getOutputCol(),
                      outputCol="features")


lr = LogisticRegression(maxIter=10)

pipeline = Pipeline(stages=[tokenizer, hashingTF, lr])
```

Parallel Hyperparameter Training

Spark CrossValidation Sample Code

- ParamGrid is a grid of different parameters to plug into our Pipeline segments from before
- CrossValidator is a wrapper around the pipeline it gets passed, and executes each pipeline with the values from the ParameterGrid
- The Evaluator parameter is the function we use to measure the loss of each model
- numFolds is how much we want to partition the dataset
- cvModel is our best model result from the training.
- cvModel.bestModel is an alias

```
paramGrid = ParamGridBuilder() \
    .addGrid(hashingTF.numFeatures, [10, 100, 1000]) \
    .addGrid(lr.regParam, [0.1, 0.01]) \
    .build()

crossval = CrossValidator(
        estimator=pipeline, estimatorParamMaps=paramGrid,
        evaluator=BinaryClassificationEvaluator(),
        numFolds=2)  # use 3+ folds in practice

cvModel = crossval.fit(training)
```

Parallel Hyperparameter Training

Spark CrossValidation Sample Code

- The test dataset is simply an unlabeled dataset with strings similar to the training dataset
- Predictions are generated as a new column by running transform on the test dataset
- This adds the predicted values and their probability as a new column
- Lastly, the code selects and prints several rows to show the behavior of the code

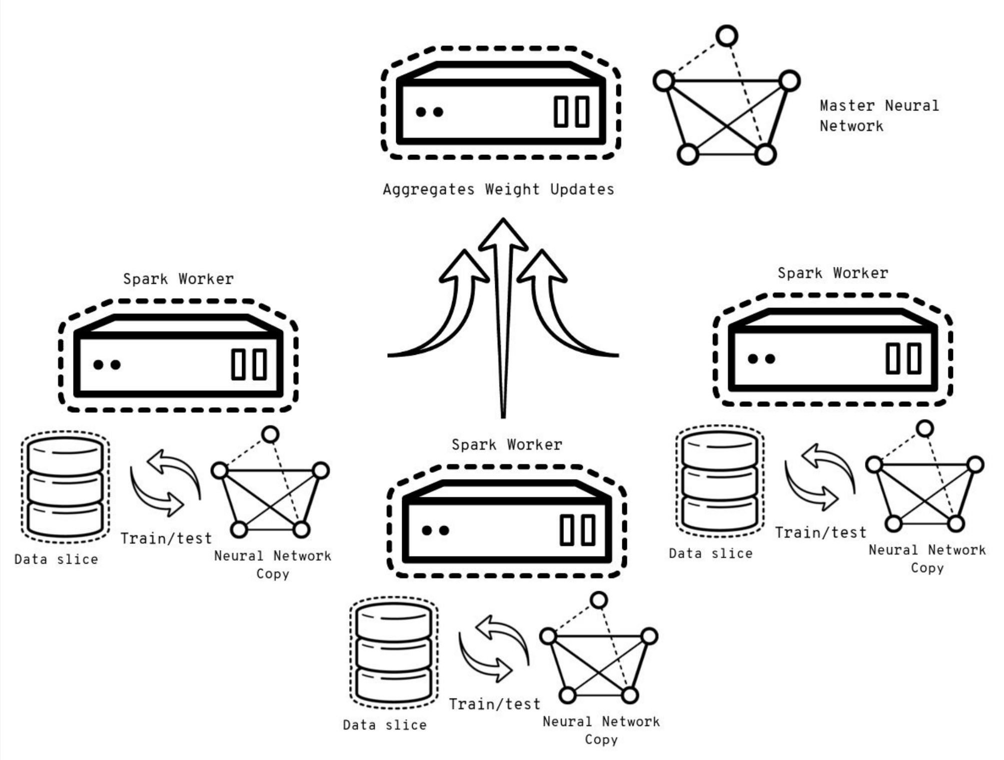
```
test = spark.createDataFrame([
    (4, "spark i j k"),
    (5, "l m n"),
    ...
], ["id", "text"])

prediction = cvModel.transform(test)

selected = prediction.select("id", "text", "probability",
                             "prediction")

for row in selected.collect():
    print(row)
```

Sparkflow Method

Alternative Parallel Training Methodology

Parameter Server with Replicated Models

- The master node runs as a parameter server
- The executor nodes all run copies of the TensorFlow graph
- After a specified number of iterations, they aggregate the weight updates to the graph back on the master node



Alternative Parallel Training Model

Sparkflow Method Sample Code

Straight off github:
https://github.com/lifeomic/sparkflow

```
from pyspark.sql import SparkSession
from sparkflow.graph_utils import build_graph
from sparkflow.tensorflow_async import SparkAsyncDL
import tensorflow as tf
from pyspark.ml.feature import VectorAssembler, OneHotEncoder
from pyspark.ml.pipeline import Pipeline

spark =
SparkSession.builder.appName("SparkflowMNIST").getOrCreate()

def small_model():
    x = tf.placeholder(tf.float32, shape=[None, 784], name='x')
    y = tf.placeholder(tf.float32, shape=[None, 10], name='y')
    layer1 = tf.layers.dense(x, 256, activation=tf.nn.relu)
    layer2 = tf.layers.dense(layer1, 256,
activation=tf.nn.relu)
    out = tf.layers.dense(layer2, 10)
    z = tf.argmax(out, 1, name='out')
    loss = tf.losses.softmax_cross_entropy(y, out)
    return loss
```

```
df = spark.read.option("inferSchema",
"true").csv('mnist_train.csv')
mg = build_graph(small_model)

va = VectorAssembler(inputCols=df.columns[1:785],
outputCol='features')
encoded = OneHotEncoder(inputCol='_c0', outputCol='labels',
dropLast=False)

spark_model = SparkAsyncDL(
    inputCol='features',
    tensorflowGraph=mg,
    tfInput='x:0',
    tfLabel='y:0',
    tfOutput='out:0',
    tfLearningRate=.001,
    iters=20,
    predictionCol='predicted',
    labelCol='labels',
    verbose=1
)

p = Pipeline(stages=[va, encoded, spark_model]).fit(df)
p.write().overwrite().save("location")
```

Image retrieved from
https://chatbotslife.com/training-mxnet-part-1-mnist-6f0dc4210c62

MNIST

For reference, an example of the MNIST dataset

- MNIST for reference is usually one of these kinds of datasets containing images of handwritten digits
- In the example code, it's been transformed into a CSV

Alternative Parallel Training Model

Sparkflow Method Deeper Dive

- This code is plain tensorflow
- A good option when your main skillset is tensorflow
- The function returns the loss metric to be minimized
- The rest of the model is optimized later on in the code

```
import tensorflow as tf

def small_model():
    x = tf.placeholder(tf.float32, shape=[None, 784], name='x')
    y = tf.placeholder(tf.float32, shape=[None, 10], name='y')
    layer1 = tf.layers.dense(x, 256, activation=tf.nn.relu)
    layer2 = tf.layers.dense(layer1, 256, activation=tf.nn.relu)
    out = tf.layers.dense(layer2, 10)
    z = tf.argmax(out, 1, name='out')
    loss = tf.losses.softmax_cross_entropy(y, out)
    return loss
```


redhat.

Alternative Parallel Training Model

Sparkflow Method Deeper Dive

- spark.read pulls the MNIST in CSV format into a spark dataframe. Note the inferSchema bit, since the data needs to be interpreted as integers not strings (the default)

```
from sparkflow.graph_utils import build_graph
from pyspark.ml.feature import VectorAssembler, OneHotEncoder

df = spark.read.option("inferSchema", "true").csv(
                                                  'swift://testdata/mnist_train.csv')
```

- build_graph builds the actual graph and serializes it to reside on the parameter server. It takes our small_model function from earlier

```
mg = build_graph(small_model)
```

- The VectorAssembler does the cleaning of the input columns into feature vectors
- Finally it sets up a one-hot encoder pipeline stage

```
#Assemble and one hot encode
va = VectorAssembler(inputCols=df.columns[1:785],
                     outputCol='features')

encoded = OneHotEncoder(inputCol='_c0', outputCol='labels',
                        dropLast=False)
```

Alternative Parallel Training Model

Sparkflow Method Deeper Dive

- **SparkAsyncDL** is the major piece of this code. It creates the parameter server, replicates the graph, and instructs the nodes to share updates
- The pipeline step creates the regular spark pipeline and applies our vectorizer, encoder, and tensorflow model to the data
- The last step just saves off the model
- Note that this doesn't optimize the learning rate or other hyperparameters automatically

```
from sparkflow.tensorflow_async import SparkAsyncDL
from pyspark.ml.pipeline import Pipeline

spark_model = SparkAsyncDL(
    inputCol='features',
    tensorflowGraph=mg,
    tfInput='x:0',
    tfLabel='y:0',
    tfOutput='out:0',
    tfLearningRate=.001,
    iters=20,
    predictionCol='predicted',
    labelCol='labels',
    verbose=1
)

p = Pipeline(stages=[va, encoded, spark_model]).fit(df)
p.write().overwrite().save("location")
```

THANK YOU

plus.google.com/+RedHat

facebook.com/redhatinc

linkedin.com/company/red-hat

twitter.com/RedHatNews

youtube.com/user/RedHatVideos