

Byzer-lang_Cifar10_DeepLearning_Demo

这篇会分四个部分。

1. 安装pip依赖库
2. 准备Cifar10 图片数据集
3. 分布式对图片预处理
4. 使用Tensorflow进行分布式模型训练
5. 将模型转化为UDF函数，从而能够应用在批，流，API等场景中

安装pip依赖库

请严格按照如下版本安装依赖库（重点关注aiohttp的版本为3.7.4），同时需要在 Byzer Driver 侧和 Ray 侧安装

Apache

```
1 pyarrow==4.0.1
2 ray[default]==1.8.0
3 aiohttp==3.7.4
4 pandas>=1.0.5; python_version < '3.7'
5 pandas>=1.2.0; python_version >= '3.7'
6 requests
7 matplotlib~=3.3.4
8 uuid~=1.30
9 pyjava
10 opencv-python
```

可使用如下命令安装：

Apache

```
1 # 创建独立的ray环境
2 conda create --name ray1.8.0 python=3.6.13
3 # 使用环境ray1.8.0
4 conda activate ray1.8.0
5 # 安装依赖
6 pip install pyarrow==4.0.1 "ray[default]==1.9.0" aiohttp==3.7.4 "pandas>=1.0.5" requests "matplotlib~=3.3.4" "uuid~=1.30" pyjava opencv-python
```

准备Cifar10 图片数据集

桌面版默认集成了mysql-shell插件。如果你是自己部署的 Byzer Notebook 以及 Byzer-engine，那么需要通过如下方式安装插件：

C#

```
1 !plugin app remove "mysql-shell-3.0";  
2 !plugin app add - "mysql-shell-3.0";
```

在拥有了如上插件后，我们就是使用功能 `!sh` 执行shell命令了。

CSS

```
1 !sh wget "https://github.com/allwefantasy/spark-deep-learning-toy/releases/download/v0.01/cifar.tgz";
```

通过上面的指令下载好图片后，接着进行解压，然后拷贝到我们的引擎的存储上：

Groovy

```
1 !sh mkdir -p /tmp/cifar10 /tmp/cifar10raw;  
2 !sh tar -xf "cifar.tgz" "-C" "/tmp/cifar10raw";  
3 !copyFromLocal /tmp/cifar10raw /tmp/cifar10;
```

此时，如果你使用如下命令应该就可以查看到数据集：

Ruby

```
1 !hdfs -ls /tmp/;
```

分布式对图片预处理

该部分代码会使用Python脚本，同时需要用户安装有Ray。请务必查看 [Byzer-python 指南](#) 来了解相关配置。

我们先来加载图片，加载方式比较简单，我们按二进制文件来进行加载（也可以按图片加载）

Ruby

```
1 load binaryFile.`/tmp/cifar10/cifar/train/*.png` as cifar10;
```

如果执行过程中出现如下异常，详情如下：

C#

```
1 Packet for query is too large (22712770 > 4194304). You can change this value  
on the server by setting the max_allowed_packet' variable.
```

请添加如下 MySQL 参数(也可以在 MySQL 命令行中动态配置)

C#

```
1 [mysqld]  
2 max_allowed_packet=200M
```

考虑到图片加载是一个比较费时的操作，同时我们要控制并行度，所以我们先重新设置下并行度，然后在保存到数据湖里：

Ruby

```
1 run cifar10 as TableRepartition.`` where partitionNum="4" as newCifar10;  
2 save overwrite newCifar10 as delta.`data.raw_cifar10` where mergeSchema="true"  
;
```

现在可以看加载下数据湖里的数据，并且统计下图片数量：

SQL

```
1 load delta.`data.raw_cifar10` as raw_cifar10_table;
2 select count(*) from raw_cifar10_table as output;
```

统计结果应该是五万条数据。

现在，我们希望把图片转化为28*28的大小，这个时候我们可以利用open-cv来完成：

Python

```
1  %%python
2  %%input=raw_cifar10_table
3  %%output=cifar10_resize
4  %%cache=true
5  %%schema=st(field(content,binary),field(path,string))
6  %%dataMode=data
7  %%env=source /opt/miniconda3/bin/activate ray1.8.0
8
9
10 from pyjava.api.mlsql import RayContext
11
12 ray_context = RayContext.connect(globals(),"127.0.0.1:10001")
13
14 def resize_image(row):
15     import io,cv2,numpy as np
16     new_row = {}
17     image_bin = row["content"]
18     oriimg = cv2.imdecode(np.frombuffer(io.BytesIO(image_bin).getbuffer(),np.uint8),1)
19     newimage = cv2.resize(oriimg,(28,28))
20     is_success, buffer = cv2.imencode(".png", newimage)
21     io_buf = io.BytesIO(buffer)
22     new_row["content"]=io_buf.getvalue()
23     new_row["path"]= row["path"]
24     return new_row
25
26 ray_context.foreach(resize_image)
```

上面的代码，我们连接一个本地Ray集群（127.0.0.1:10001），并确保Ray集群安装了 `opencv-python`。接着，我们定义了一个 `resize_image` 方法，该方法会被每条记录回调，从而实现图片的处理。实际上，用户完全可以用Ray API自己完成这些工作，但是Byzer提供了良好的API方便大家做相关的处理。

现在，我们得到了一张表 `cifar10_resize`，现在开心的保存到数据湖里去：

Apache

```
1 save overwrite cifar10_resize as delta.`data.cifar10x28x28`;
```

为了方便获取数组最后一个元数，我们定义了新的UDF `arrayLast` 函数：

Python

```
1 register ScriptUDF.`` as arrayLast where
2 lang="scala"
3 and code='''def apply(a:Seq[String])={
4     a.last
5 }'''
6 and udfType="udf";
```

我们对从路径抽取出文件名，最后，我们把二进制文件还原为图片文件保存到文件系统里去。

C#

```
1 select arrayLast(split(path,"/")) as fileName,content
2 from cifar10_resize
3 as final_dataset;
4
5 save overwrite final_dataset as image.`/tmp/size-28x28`
6 where imageColumn="content"
7 and fileName="fileName";
```

使用 Tensorflow 进行分布式训练

上面，我们把五万张图片缩放为 28*28 的规格。现在，我们要加载这些图片，因为要反复调试，所以我们将加载后的结果保存到数据湖里，方便后续反复使用。

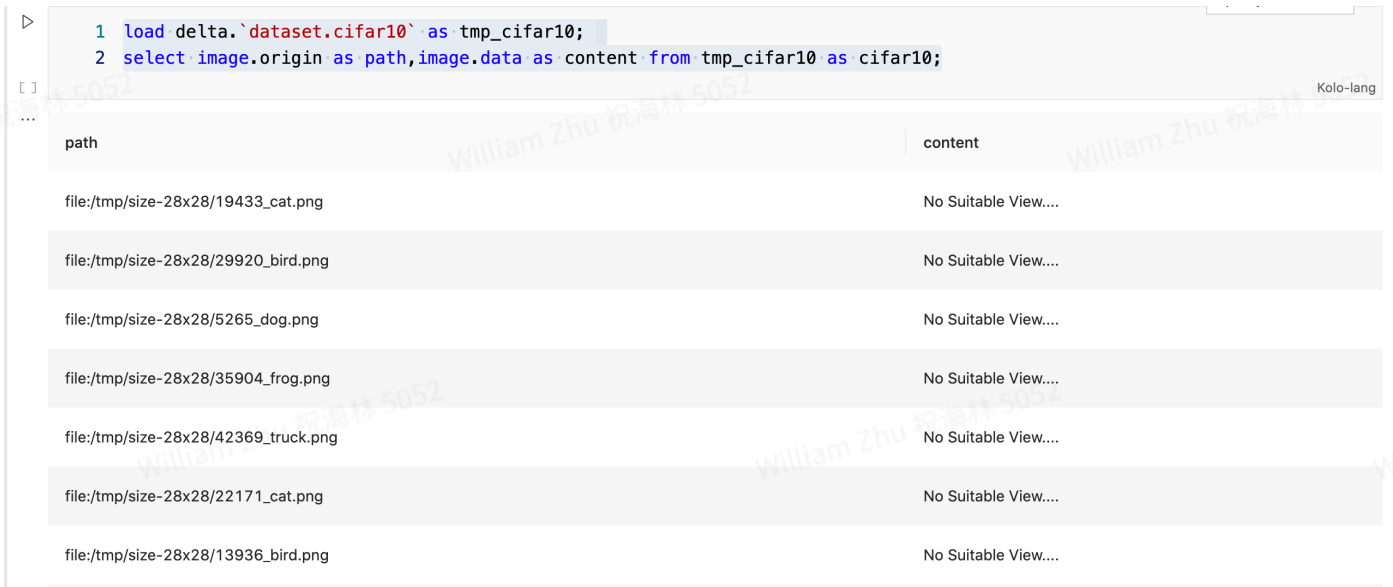
Haskell

```
1 -- 加载 ResizeImage.mlsqLnb 已经预处理好的图片数据
2 -- 因为涉及到 list 目录，所以比较慢，我们可以把加载的结果保存下
3 load image.`/tmp/size-28x28/*.png` where implClass="image" as cifar10;
4 save overwrite cifar10 as delta.`dataset.cifar10` where fileNum="4" and overwriteSchema="true";
```

加载图片看下结果：

SQL

```
1 load delta.`dataset.cifar10` as tmp_cifar10;
2 select image.origin as path, image.data as content from tmp_cifar10 as cifar10;
```



The screenshot shows a SQL query execution interface. The query is:

```
1 load delta.`dataset.cifar10` as tmp_cifar10;
2 select image.origin as path, image.data as content from tmp_cifar10 as cifar10;
```

The results are displayed in a table with two columns: 'path' and 'content'. The 'content' column shows 'No Suitable View...' for all rows.

path	content
file:/tmp/size-28x28/19433_cat.png	No Suitable View...
file:/tmp/size-28x28/29920_bird.png	No Suitable View...
file:/tmp/size-28x28/5265_dog.png	No Suitable View...
file:/tmp/size-28x28/35904_frog.png	No Suitable View...
file:/tmp/size-28x28/42369_truck.png	No Suitable View...
file:/tmp/size-28x28/22171_cat.png	No Suitable View...
file:/tmp/size-28x28/13936_bird.png	No Suitable View...

接下来，我们要从路径中抽取分类，并且把分类转化为数字：

C#

```
1 -- extract category from path
2 select split(arrayLast(split(path,"/")),"_")[1] as labelName,content
3 from cifar10
4 as tmp_cifar10;
5
6 -- mapping category to index (0-9)
7 train tmp_cifar10 as StringIndex.`/tmp/labelToIndex` where inputCol="labelName"
8 and outputCol="label"
9 as final_cifar10;
```

现在，可以利用Tensorflow库进行分布式训练了：

Python

```
1  %%python
2  %%input=final_cifar10
3  %%output=cifar10_model
4  %%cache=true
5  %%schema=file
6  %%dataMode=model
7  %%env=source /opt/miniconda3/bin/activate ray1.8.0
8
9  from functools import reduce
10 import os
11 import ray
12 import numpy as np
13 from tensorflow.keras import models, layers
14 from tensorflow.keras import utils as np_utils
15 from pyjava.api.mlsql import RayContext
16 from pyjava.storage import streaming_tar
17
18
19 ray_context = RayContext.connect(globals(), "127.0.0.1:10001")
20 data_servers = ray_context.data_servers()
21 replica_num = len(data_servers)
22 print(f"total workers {replica_num}")
23 def data_partition_creator(data_server):
24     temp_data = [item for item in RayContext.collect_from([data_server])]
25
26     train_images = np.array([np.array(list(item["content"])) for item in temp_data])
```

```

27     train_labels = np_utils.to_categorical(np.array([item["label"] for item in
temp_data]))
28     train_images = train_images.reshape((len(temp_data),28*28*3))
29     return train_images,train_labels
30
31 def create_tf_model():
32     network = models.Sequential()
33     network.add(layers.Dense(512,activation="relu",input_shape=(28*28*3,)))
34     network.add(layers.Dense(10,activation="softmax"))
35     network.compile(optimizer="sgd",loss="categorical_crossentropy",metrics=[
"accuracy"])
36     return network
37
38 @ray.remote
39 class Network(object):
40     def __init__(self,data_server):
41         self.model = create_tf_model()
42         # you can also save the data to local disk if the data is
43         # not fit in memory
44         self.train_images,self.train_labels = data_partition_creator(data_serv
er)
45
46
47     def train(self):
48         history = self.model.fit(self.train_images,self.train_labels,batch_siz
e=128)
49         return history.history
50
51     def get_weights(self):
52         return self.model.get_weights()
53
54     def set_weights(self, weights):
55         # Note that for simplicity this does not handle the optimizer state.
56         self.model.set_weights(weights)
57
58     def get_final_model(self):
59         model_path = os.path.join("/", "tmp", "minist_model")
60         self.model.save(model_path)
61         model_binary = [item for item in streaming_tar.build_rows_from_file(mo
del_path)]
62         return model_binary
63     def shutdown(self):
64         ray.actor.exit_actor()
65
66 workers = [Network.remote(data_server) for data_server in data_servers]
67 ray.get([worker.train.remote() for worker in workers])
68 _weights = ray.get([worker.get_weights.remote() for worker in workers])

```



```

69
70 def epoch_train(weights):
71     sum_weights = reduce(lambda a,b: [(a1 + b1) for a1,b1 in zip(a,b)],weights
72     s)
73     averaged_weights = [layer/replica_num for layer in sum_weights]
74     ray.get([worker.set_weights.remote(averaged_weights) for worker in worker
75     s])
76     ray.get([worker.train.remote() for worker in workers])
77     return ray.get([worker.get_weights.remote() for worker in workers])
78
79
80 for epoch in range(6):
81     _weights = epoch_train(_weights)
82
83 model_binary = ray.get(workers[0].get_final_model.remote())
84 [worker.shutdown.remote() for worker in workers]
85 ray_context.build_result(model_binary)

```

请确保你在Ray集群里安装了 tensorflow 。

我们仔细讲解下上面的Python代码。首先，

Makefile

```

1 data_servers = ray_context.data_servers()

```

通过上面这段代码，我们可以拿到一个数据集的切片引用。比如在我们当前的例子里，有四个切片，也就是一个数据集被分成了四份，每一份都可以通过下面的代码来获取：

CSS

```

1 RayContext.collect_from([data_server])

```

我们定义了data_partition_creator函数，该函数用来获取数据切片的数据以及转化为tensorflow能认的格式。create_tf_model函数则是创建一个神经网络。他们都会在 class Network中被使用，而Network是remote类，这意味着他们会在不同的进程中被初始化。

JavaScript

```
1 workers = [Network.remote(data_server) for data_server in data_servers]
2 ray.get([worker.train.remote() for worker in workers])
```

上面两行代码中的第一行，我们新建了四个Network实例，这四个network实例会处于Ray集群的不同Python进程中。创建完实例后，我们调用实例的train方法进行训练。通过ray.get 来进行等待。这样我们就得到了第一轮参数：

JavaScript

```
1 _weights = ray.get([worker.get_weights.remote() for worker in workers])
```

我们将这四组参数加权平均，作为下一轮的基准参数。这本质上相当于自己用Ray实现了一个Parameter Server。对应的代码在 epoch_train 函数里。最后，我们将模型以二进制流返回给引擎。

Python代码执行完成后，我们会把二进制流以目录结构的方式展示给用户：

```
78 | _weights = epoch_train(_weights)
79 |
80 | model_binary = ray.get(workers[0].get_final_model.remote())
81 | [worker.shutdown.remote() for worker in workers]
82 | ray_context.build_result(model_binary)
```



现在，我们把模型保存到数据湖里去：

JavaScript

```
1 save overwrite cifar10_model as delta.`ai_model.cifar_model`;
```

至此，我们以Parameter Server的模式完成了模型的分布式训练。

模型部署

一般模型都要部署到三个场景中：

1. 批
2. 流
3. API

我们通过将模型转化为UDF来完成模型的全景部署。

首先，加载模型：

SQL

```
1 load delta.`ai_model.cifar_model` as cifar_model;
```

接着，我们将模型注册成UDF函数：

Python

```
1 !python conf "rayAddress=127.0.0.1:10001";
2 !python conf "schema=file";
3 !python env "PYTHON_ENV=source /opt/miniconda3/bin/activate ray1.8.0";
4 !python conf "dataMode=model";
5 !python conf "runIn=driver";
6
7 register Ray.`cifar_model` as model_predict where
8 maxConcurrency="2"
9 and debugMode="true"
10 and registerCode=''
11
12 import ray
13 import numpy as np
14 from pyjava.api.mlsql import RayContext
15 from pyjava.udf import UDFMaster,UDFWorker,UDFBuilder,UDFBuildInFunc
16
17 ray_context = RayContext.connect(globals(), context.conf["rayAddress"])
18
19 def predict_func(model,v):
20     train_images = np.array([v])
21     train_images = train_images.reshape((1,28*28*3))
22     predictions = model.predict(train_images)
23     return {"value":[[float(np.argmax(item)) for item in predictions]]}
24
25 UDFBuilder.build(ray_context,UDFBuildInFunc.init_tf,predict_func)
26
27 ''' and
28 predictCode=''
29
30 import ray
31 from pyjava.api.mlsql import RayContext
32 from pyjava.udf import UDFMaster,UDFWorker,UDFBuilder,UDFBuildInFunc
33
34 ray_context = RayContext.connect(globals(), context.conf["rayAddress"])
35 UDFBuilder.apply(ray_context)
36
37 '''
38 ;
```

registerCode 部分，本质上我们只要提供一个预测函数即可，该函数系统会传递两个参数进来，一个是模型，一个是数据。用户要完成如何将两者结合的的逻辑。第二段代码predictCode则完全是模板代码，复制黏贴就可以。

其中值得注意的是，

Python

```
1 maxConcurrency="2"
```

该参数配置了请求并发度。这无论在批还是流，或者API服务里都很重要。如果要满足不通场景，用户可以注册多次从而分别给不同场景提供合理的并发能力。

现在我们得到了一个函数叫`model_predict`，他可以对图片进行分类预测了。

不过考虑到我们需要把binary转化为无符号数字数组，所以我们需要有一个函数完成这个事情，所以我们可以创建一个叫byteArrayToUnsignedIntArray 函数。

Python

```
1 register ScriptUDF.` ` as byteArrayToUnsignedIntArray
2 where lang="scala"
3 and code='''def apply(a:Array[Byte])={
4     a.map(_ & 0xFF).map(_.toDouble).toArray
5 }'''
6 and udfType="udf";
```

现在，可以做预测了：

SQL

```
1 select model_predict(array(byteArrayToUnsignedIntArray(content))) from final_c
  ifar10 limit 10 as output;
```

输出结果如下：

```
1 select model_predict(array(byteArrayToUnsignedIntArray(content))) from final_cifar10 limit 10 as output;
```

model_predict(array(UDF(content)))

[[5]]

[[5]]

[[5]]

总结

通过上面这个案例，通过使用Byzer-lang,我们仅仅需要少量的python代码，就可以完成较为复杂的数据加载，汇总，预处理，分布式训练，模型部署等整个AI的pipeline。SQL和Python之间无缝衔接，并且无论SQL还是Python都实现了分布式，这个能力是非常强大的。