Flink-AI-Extended Usage Report

在之前的一到两周中，我先是在纯TensorFlow环境下中实现了一个生成式文档摘要应用，能从文件中读取文章和摘要进行训练，或是从文件中读取文章并生成摘要输出到文件。接下来通过Flink-AI-Extended封装并调用TF模型的训练/引用过程，对模型的输入、输出以及超参设置均在Flink层实现并连接到TF。至此，我根据下述的目标，以用户和开发者的角度，尝试提出一些在使用过程中觉得Flink-AI-Extended可能存在的问题，并尽量提出相应的解决方向或方案。

Motivation

1. 若作为Flink-AI-Extended的用户，在使用Flink-AI-Extended时有哪些地方是感到迷惑、不便的，会期待以什么样的方式进行什么样的操作?
2. 若作为Flink-AI-Extended的开发者，应该如何引导用户以便捷正确的方式开发他们的TF-on-Flink程序，有哪些地方是做得不够的，现有的实现方案是否很好地满足所有开发场景?
3. 若发现某些问题，我会提出什么样的解决方案，以满足什么样的目标?

Issues

1. Python中的device、session配置过程需要指引

在Flink上调用TF程序最简单的方式是在Java中配置好相应的zookeeper server，直接调用python文件中的某个函数：

```
//python file to execute
String script = "./add.py";
StreamExecutionEnvironment streamEnv =
StreamExecutionEnvironment.getExecutionEnvironment();
Map<String, String> prop = new HashMap<>();
prop.put(MLConstants.CONFIG_STORAGE_TYPE, MLConstants.STORAGE_ZOOKEEPER);
prop.put(MLConstants.CONFIG_ZOOKEEPER_CONNECT_STR, "localhost:2181");
//TFConfig(workerNum, psNum, properties, pythonFiles, funName, envPath)
TFConfig config = new TFConfig(2, 1, prop, script, "map_func", null);
//call function
TFUtils.train(streamEnv, null, config);
//truly execute
streamEnv.execute();
```

然而，在python上也许进行相应的配置才能实现TF与Flink的连接，这部分的配置稍微复杂了一些，也缺乏详细的文档和说明，对于之前没有使用过TF分布式训练的用户来说可能比较迷惑：

```
def map_func(context):
    tf_context = TFContext(context)
    job_name = tf_context.get_role_name()
    index = tf_context.get_index()
```

```
    cluster_json = tf_context.get_tf_cluster()

    cluster = tf.train.ClusterSpec(cluster=cluster_json)
    server = tf.train.Server(cluster, job_name=job_name, task_index=index)
    sess_config = tf.ConfigProto(allow_soft_placement=True,
log_device_placement=False,
                                 device_filters=["/job:ps",
"/job:worker/task:%d" % index])
    t = time.time()
    if 'ps' == job_name:
        from time import sleep
        while True:
            sleep(1)
    else:
        with
tf.device(tf.train.replica_device_setter(worker_device='/job:worker/task:' +
str(index), cluster=cluster)):
            train_ops = build_graph()
            hooks = [tf.train.StopAtStepHook(last_step=2)]
            with tf.train.MonitoredTrainingSession(master=server.target,
config=sess_config,

 checkpoint_dir="./target/tmp/s1/" + str(t),
                                                hooks=hooks) as mon_sess:
                while not mon_sess.should_stop():
                    print (mon_sess.run(train_ops, feed_dict={a: [1.0, 2.0,
3.0]}))
                    sys.stdout.flush()
```

比如说，上面的例子中，sess_config之前都是从context中加载信息并创建一些对象，这一部分都是相对固定的但如果让用户自己来写却不容易。同样的，tf.device的配置也相对固定且不容易自己写。而session配置方面则与一般单机TF程序不同，用户可能需要被指导在分布式环境下，何时采用什么样的Session、配置什么样的参数。比如说在训练时可用MonitoredTrainingSession，在引用时可使用一般的MonitoredSession，worker-0需要使用ChiefSessionCreator，其他worker使用WorkerSessionCreator等。其实对用户来说，真正关心的可能只有session的配置，因为这也是他们改写TF程序时需要重点关注的地方，我觉得可以把其他的配置过程封装起来，让用户把注意力放在他们最需要注意的地方。以下是我尝试改进后的代码：

```
def map_func(context):
    # Singleton Mode, get TFContext by get_or_create(None), instantiated by
get_or_create(context) once only, raise error if instantiated again
    tf_context = TFContext.get_or_create(context)
    if 'ps' == tf_context.get_role_name():
        from time import sleep
        while True:
            sleep(1)
    else:
        with flink_server_device(tf_context) as device, server, sess_config:
```

```
            train_ops = build_graph()
            hooks = [tf.train.StopAtStepHook(last_step=2)]
            with tf.train.MonitoredTrainingSession(master=server.target,
config=sess_config,

 checkpoint_dir="./target/tmp/s1/" + str(t),
                                                   hooks=hooks) as mon_sess:
                while not mon_sess.should_stop():
                    print (mon_sess.run(train_ops, feed_dict={a: [1.0, 2.0,
3.0]}))
                    sys.stdout.flush()
```

```
def flink_cluster_device(tf_context):
    index = tf_context.get_index()
    cluster_json = tf_context.get_tf_cluster()
    cluster = tf.train.ClusterSpec(cluster=cluster_json)

    server = tf.train.Server(cluster, job_name=job_name, task_index=index)
    device =
tf.device(tf.train.replica_device_setter(worker_device='/job:worker/task:' +
str(index), cluster=cluster))
    sess_config = tf.ConfigProto(allow_soft_placement=True,
log_device_placement=False,
                                 device_filters=["/job:ps",
"/job:worker/task:%d" % index])
    return device, server, sess_config
```

我提出的想法是，在一开始，将java传过来的context实例化一个TFContext，并采用单例模式以供全局访问。然后在配置device部分调用一个封装好的相对固定的函数flink_cluster_device，并返回创建session所必要的信息server和sess_config。这样的话就可以用两句简单且容易理解的语句完成大部分与模型无关的配置工作，让用户专注于模型的构建和session的配置。同时，如果可以在文档中加入一些关于分布式环境下Session相关的介绍，可以让用户更方便地进行改写和迁移。

2. TF与Flink之间的输入、输出、参数传递过程渴望指引

只是在Flink上单纯地调用TF程序显然是不够的，用户使用Flink-AI-Extended的目的无非是想获得Flink流处理能力和易扩展性的加持，因此，如何把Flink的数据流入TF程序并把结果流回Flink肯定是用户关注的重中之重。然而现在的文档貌似缺乏这一部分的说明和指导，只能参考源码样例进行摸索，且使用起来相对繁琐，对用户来说可能存在一定的试错成本。如果能提供一套简单易用的数据传输配置接口以及相应的文档说明，用户就能方便地真正打通flink与tf并部署实际的应用。

现阶段我了解的数据传输流程是：

- **Flink编码**：Flink对每一条entry（比如Row）根据指定的格式进行编码（如Csv、tf.Example、ByteArray等），传送给TF
- **TF解码**：TF对entry进行解码，转换成input_tensor
- **TF运算**：TF把input_tensor输入模型进行计算，得到output_tensor

- **TF编码**：TF根据指定格式对output_tensor进行编码，传送给Flink
- **Flink解码**：Flink对output_tensor进行解码，得到entry对应的结果

根据模型的不同，可以省去input部分或output部分，或者两者都省去。

在Java上，主要需要配置Flink编码与Flink解码相关的信息，通过往TFConfig中添加4个properties来完成。以下面代码为例，输入为一个String，字段名"article"，输出也是一个String，字段名"abstract"，编码格式为tf.Example，配置过程如下：

```
private static void setExampleCodingTypeRow(TFConfig config) {
    String[] namesInput = {"article"};
    DataTypes[] typesInput = {DataTypes.STRING};
    String strInput = ExampleCodingConfig
      .createExampleConfigStr(namesInput, typesInput,
                               ExampleCodingConfig.ObjectType.ROW, Row.class);
    LOG.info("input if example config: " + strInput);

    String[] namesOutput = {"abstract"};
    DataTypes[] typesOutput = {DataTypes.STRING};
    String strOutput = ExampleCodingConfig
      .createExampleConfigStr(namesOutput, typesOutput,
                               ExampleCodingConfig.ObjectType.ROW, Row.class);
    LOG.info("output if example config: " + strOutput);

    config.getProperties().put(TFConstants.INPUT_TF_EXAMPLE_CONFIG, strInput);
    config.getProperties().put(TFConstants.OUTPUT_TF_EXAMPLE_CONFIG,
strOutput);
    config.getProperties().put(MLConstants.ENCODING_CLASS,
                               ExampleCoding.class.getCanonicalName());
    config.getProperties().put(MLConstants.DECODING_CLASS,
                               ExampleCoding.class.getCanonicalName());
}
```

现在来说，上面Java这部分的配置仍需要用户自己来写，其实对用户来说，输入和输出的schema一般都是明确的，我们完全可以把这部分封装，根据schema来自动配置编码解码格式。下面是我写的一个工具函数：

```
    /***
     * Map org.apache.flink.api.common.typeinfo.TypeInformation to
     * com.alibaba.flink.ml.operator.util.DataTypes
     * @throws RuntimeException when TypeInformation is of unsupported type
     */
    public static DataTypes typeInformationToDataTypes(TypeInformation
typeInformation)
      throws RuntimeException {
        if (typeInformation == BasicTypeInfo.STRING_TYPE_INFO) {
            return DataTypes.STRING;
        }
        //....
```

```
        else {
            throw new RuntimeException("Unsupported data type of " +
                                       typeInformation.toString());
        }
    }
```

```
    public static void configureExampleCoding(TFConfig config,
                                              String[] encodeNames,
                                              TypeInformation[] encodeTypes,
                                              String[] decodeNames,
                                              TypeInformation[] decodeTypes,
                                              ExampleCodingConfig.ObjectType
entryType,
                                              Class entryClass) throws
RuntimeException {
        DataTypes[] encodeDataTypes = Arrays
                .stream(encodeTypes)
                .map(Utils::typeInformationToDataTypes)
                .toArray(DataTypes[]::new);
        String strInput = ExampleCodingConfig
          .createExampleConfigStr(encodeNames, encodeDataTypes, entryType,
entryClass);
        LOG.info("input if example config: " + strInput);

        DataTypes[] decodeDataTypes = Arrays
                .stream(decodeTypes)
                .map(Utils::typeInformationToDataTypes)
                .toArray(DataTypes[]::new);
        String strOutput = ExampleCodingConfig
          .createExampleConfigStr(decodeNames, decodeDataTypes, entryType,
entryClass);
        LOG.info("output if example config: " + strOutput);

        config.getProperties().put(TFConstants.OUTPUT_TF_EXAMPLE_CONFIG,
strOutput);
        config.getProperties().put(TFConstants.INPUT_TF_EXAMPLE_CONFIG,
strInput);
        config.getProperties().put(MLConstants.ENCODING_CLASS,
                                   ExampleCoding.class.getCanonicalName());
        config.getProperties().put(MLConstants.DECODING_CLASS,
                                   ExampleCoding.class.getCanonicalName());
    }

    public static void configureExampleCoding(TFConfig config,
                                              TableSchema encodeSchema,
                                              TableSchema decodeSchema,
                                              ExampleCodingConfig.ObjectType
entryType,
```

```
                                                Class entryClass) throws
RuntimeException {
        configureExampleCoding(config,
                               encodeSchema.getFieldNames(),
                               encodeSchema.getFieldTypes(),
                               decodeSchema.getFieldNames(),
                               decodeSchema.getFieldTypes(),
                               entryType, entryClass);
    }
```

这样子，用户就能通过简单的一条语句进行编码配置：

```
Table input = tableEnv.fromDataStream(streamEnv.readTextFile(inputTableFile),
"article");
TableSchema outSchema =
  new TableSchema(new String[]{"abstract"},
                  new TypeInformation[]{BasicTypeInfo.STRING_TYPE_INFO});

Utils.configureExampleCoding(config,
                             input.getSchema(), outSchema,
                             ObjectType.ROW, Row.class);
```

Java部分之后，还需要在Python上配置TF解码与TF编码相关的信息，原来的使用方式就不再赘述，我同样的也做了一层封装。AbstractFlinkReader是用来读取Flink传过来的输入，需要实现_features和_decode两个方法，但其实是可以通过context传来的schema来自动实现的：

```
class AbstractFlinkReader(object):
    __metaclass__ = ABCMeta

    def __init__(self, context, batch_size=1):
        """
        Initialize the batch reader
        :param context: TFContext
        """
        self._context = context
        self._batch_size = batch_size
        self._build_graph()

    @abstractmethod
    def _features(self):
        # TODO: Implement this function using context automatically
        """
        Examples:
        ```
        features = {'article': tf.FixedLenFeature([], tf.string)}
        return features
        ```
```

```
        :return: A `dict` mapping feature keys to `FixedLenFeature`,
`VarLenFeature`, and `SparseFeature` values.
        """
        pass

    @abstractmethod
    def _decode(self, features):
        # TODO: Implement this function using context automatically
        """
        Each feature in features is `Tensor` of type `string`,
        this func is to decode them to `Tensor` of right type.

        Examples:
        ```
        image = tf.decode_raw(features['image_raw'], tf.uint8)
        label = tf.one_hot(features['label'], 10, on_value=1)
        article = features['article']

        return image, label, article
        ```

        :param features: A `list` of `Tensor` of type `string`
        :return: A `list` of `Tensor` of right type
        """
        pass

    def _build_graph(self):
        dataset = self._context.flink_stream_dataset()
        dataset = dataset.map(lambda record: tf.parse_single_example(record,
features=self._features()))
        dataset = dataset.map(self._decode)
        dataset = dataset.batch(self._batch_size)
        iterator = dataset.make_one_shot_iterator()
        self._next_batch = iterator.get_next()

    def next_batch(self, sess):
        """
        use session to get next batch
        :param sess: the session to execute operator
        :return:
        """
        try:
            batch = sess.run([self._next_batch])
            return batch
        except tf.errors.OutOfRangeError:
            return None
```

AbstractFlinkWriter是用来把结果写回Flink的，需要实现_example方法，但应该也是可以通过context传来的schema来自动实现的：

```
class AbstractFlinkWriter(object):
    __metaclass__ = ABCMeta

    def __init__(self, context):
        """
        Initialize the writer
        :param context: TFContext
        """
        self._context = context
        self._build_graph()

    def _build_graph(self):
        self._write_feed = tf.placeholder(dtype=tf.string)
        self.write_op, self._close_op =
self._context.output_writer_op([self._write_feed])

    @abstractmethod
    def _example(self, results):
        # TODO: Implement this function using context automatically
        """
        Encode the results tensor to `tf.train.Example`

        Examples:
        ```
        example = tf.train.Example(features=tf.train.Features(
            feature={
                'predict_label':
tf.train.Feature(int64_list=tf.train.Int64List(value=[results[0]])),
                'label_org':
tf.train.Feature(int64_list=tf.train.Int64List(value=[results[1]])),
                'abstract':
tf.train.Feature(bytes_list=tf.train.BytesList(value=[results[2]])),
            }
        ))
        return example
        ```

        :param results: the result list of `Tensor` to write into Flink
        :return: An Example.
        """
        pass

    def write_result(self, sess, results):
        """
        Encode the results tensor and write to Flink.
        """
```

```
        sess.run(self.write_op, feed_dict={self._write_feed:
self._example(results).SerializeToString()})

    def close(self, sess):
        """
        close writer
        :param sess: the session to execute operator
        """
        sess.run(self._close_op)
```

通过这两个辅助类，Python上整个数据传输的流程会变得更加清晰明了，使用起来也更加方便：

```
def map_func(context):
    tf_context = TFContext.get_or_create(context)
    if 'ps' == tf_context.get_role_name():
        from time import sleep
        while True:
            sleep(1)
    else:
        with flink_server_device(tf_context) as device, server, sess_config:
            # build graph
            reader = FlinkReader(tf_context, batch_size=3)
            model = SomeModel()
            writer = FlinkWriter(tf_context)

            with MonitoredTrainingSession(master=server.target,
config=sess_config) as sess:
                while not sess.should_stop():
                    batch = reader.next_batch(sess)
                    results = model.transform(sess, batch)
                    writer.write_results(sess, results)
                writer.close(sess)
```

3. TF与Flink之间的数据编码/解码机制可以更加完善

现在的编码/解码机制提供了多种方式以供选择，分别有ByteArrayCodingImpl、JsonCodingImpl、ExampleCoding、RowCSVCoding，但我认为暂时来说没有一种是较为完美的：ByteCoding过于原始，解码麻烦；JsonCoding会丢失类型信息（比如不能区分int16和int32）；ExampleCoding暂不支持所有TypeInformation到DataTypes的映射）；CSVCoding兼容性太差（不支持数组和复杂字符串）。

我觉得对于用户来说，提供过多的选择只会导致更多的学习成本，如果有一种较为完善的编码方式作为规范，用户应该是更乐意接受的。目前来看，我觉得最为完善的应该是ExampleCoding，但需要一些补充。

现在版本中，TF能接受的数据类型定义在*com.alibaba.flink.ml.operator.util.DataTypes*中，而Flink Table字段的数据类型定义为import *org.apache.flink.api.common.typeinfo.TypeInformation*，其基本类型定义在*BasicTypeInfo*和*BasicArrayTypeInfo*，但*DataTypes*和*TypeInformation*的基本类型并非一一对应，这可能会导致用户在**Flink编码**阶段某些数据不知道如何设定类型。解决方案也相对简单，把缺少

的类型补全并提供两者相互转换的接口即可。

```
    /***
     * Map org.apache.flink.api.common.typeinfo.TypeInformation to
     * com.alibaba.flink.ml.operator.util.DataTypes
     * @throws RuntimeException when TypeInformation is of unsupported type
     */
    public static DataTypes typeInformationToDataTypes(TypeInformation
typeInformation)
      throws RuntimeException {
        if (typeInformation == BasicTypeInfo.STRING_TYPE_INFO) {
            return DataTypes.STRING;
        }
        //....
        else {
            throw new RuntimeException("Unsupported data type of " +
                                       typeInformation.toString());
        }
    }
```

4. 如果能提供一套代码改写/迁移的规范就更好了

不管对于何种用户来说，使用Flink-AI-Extended意味着必须对TF程序进行一定的改写，对于他们而言，改得越少、改的地方越顶层，他们就会有越多的兴趣继续深入了解该项目。因此，如果我们能提供一套简单明了、可复用性高的规范/模版以供用户参考着来写，用户使用起来也会更方便。

比如我之前改写的那个样例，一般来说用户自己写的model是不希望去改动的，而对于Flink-AI-Extended项目来说，需要配置的地方主要有context、hyperparam、device、session、input、output，我把前两步在map_func一开始解决，并通过一个简单的flink_server_device封装device配置过程、一个FlinkReader提供input数据、一个FlinkWriter提供输出渠道，把所有与Flink相关、与TF无关的过程都放在顶层的map_func解决。这里我把ps判断那部分给去掉了，因为还暂不明白这部分的含义，所以不知道如何简化。

```
def map_func(context):
    tf_context = TFContext.get_or_create(context)
    with flink_server_device(tf_context) as device, server, sess_config:
        # build graph
        reader = FlinkReader(tf_context, batch_size=3)
        model = SomeModel()
        writer = FlinkWriter(tf_context)

        with MonitoredTrainingSession(master=server.target,
config=sess_config) as sess:
            while not sess.should_stop():
                batch = reader.next_batch(sess)
                results = model.transform(sess, batch)
                writer.write_results(sess, results)
            writer.close(sess)
```

5. （疑惑）Flink与TF的数据传输是否为流

在使用过程中我有一些疑惑：Flink与TF之间的数据传输底层机制是怎样的呢?

我想到的有三种可能的方案：一种是流式，Flink input table一条一条地喂数据给TF，在喂数据的同时TF也在一条条地处理计算，并即时地把结果吐回给Flink output table；一种是批式，input table完全扫一遍后全部传给TF，TF把所有数据接收完再开始处理计算，并且所有计算完成后才吐回给output table；还有一种是攒一个小批。