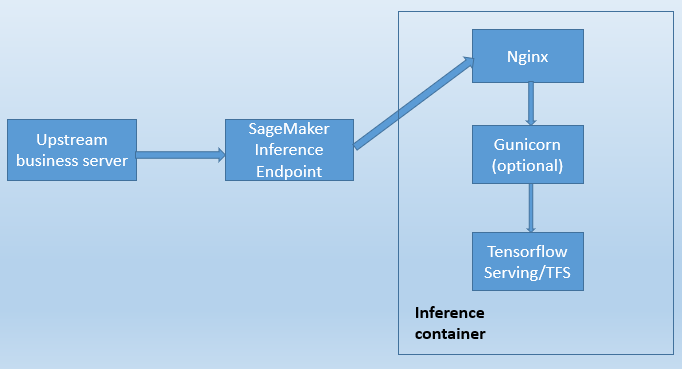
Tensorflow Serving（简称TFS）是一个很常用的模型推理开源框架，Amazon SageMaker 内建了TFS推理容器以支持TensorFlow SavedModel模型进行高效推理，Amazon SageMaker内建的TFS推理容器即可以做在线推理（比如通过SageMaker Inference Endpoint），也可以做离线推理（比如通过SageMaker Batch Transform）。本文只讨论内建的TFS的在线推理，但其实对于内建的TFS的推理加速优化和吞吐量优化的思路，离线推理场景完全可以参考。诚然，您可以通过自定义推理容器的方式来部署开源的TFS，但是一般情况下使用SageMaker内建的TFS就足够了。

接下来，我们详细的介绍一下SageMaker内建的TFS的一些细节，最后用一个真实的案例来看一下如何借助内建的TFS做推理的加速和吞吐量的提升。

1. SageMaker内建的TFS容器概览：

**内建的TFS容器本身的拓扑是**Nginx----->Gunicorn(optional)----->TFS

如果从SageMaker外面看并且使用内建TFS做在线推理的话，拓扑是这样的（这里不考虑SageMaker inference pipeline，inference pipeline的拓扑会更复杂）：



是否启用Gunicorn进程取决于是否设置了定制的 inference.py脚本用于数据预处理和后处理，或者是否启用了SageMaker Multi-Model Endpoint。

不管有没有启动Gunicorn，SageMaker发给Nginx的/ping REST API健康检查，Nginx都会把/ping转为某种TFS可以接受的REST API（比如/v1/models/${MODEL\_NAME} ）后直接发给TFS进程。

之所以拓扑如此复杂的原因是：TFS不支持/ping REST API；TFS不支持钩子函数/hook（尽管在保存模型的时候可以使用serving\_input\_receiver\_fn来让TFS在接收到请求后做数据预处理，但是没有办法做后处理）

1. SageMaker内建的TFS与推理加速和吞吐量优化相关的配置
2. Server side batch相关配置

通过设置Server side batch，可以提升吞吐量，如下表格中提到的离线推理均指SageMaker Batch Transform功能，在线推理均指SageMaker在线实时推理功能。

|  |  |  |
| --- | --- | --- |
| 环境变量 | 介绍 | 备注 |
| SAGEMAKER\_TFS\_ENABLE\_BATCHING | 开启 TFS的server side batch 功能 | 不管是否开启server side batch，TFS都支持client side batch |
| SAGEMAKER\_TFS\_BATCH\_TIMEOUT\_MICROS | TFS在合并多个客户端请求的样本为一个batch之前最多等待的时间（单位是微秒） | 如果是离线推理，建议这个timeout设置大一些（用CPU推理的话，可以设置十毫秒左右；用GPU推理的话，可以设置几毫秒）；  如果是在线推理，建议设置这个timeout为毫秒级别（比如1ms） |
| SAGEMAKER\_TFS\_MAX\_BATCH\_SIZE | TFS在合并后得到的一个batch能包含的最大的样本数量 | 对于离线推理，建议设置比较大的max batch size，只要不发生OOM；  对于在线推理，max batch size的设置和很多因素有关：是否是GPU实例，单个推理实例接受到的并发请求中包含的样本数量，高分位数延迟（比如P90）的要求，RAM或者GPU显存的可用容量，模型对单个样本的纯推理时间等。 |
| SAGEMAKER\_TFS\_NUM\_BATCH\_THREADS | 并行执行多个batch推理的最大数量 | 这个数量一般设置为推理实例的VCPU的数量或者物理core的数量。  对于CPU实例，batch推理可能会并行；对于GPU推理，这里的batch推理只能串行执行（对于GPU推理，也可以尝试把这个值设置为 1） |
| SAGEMAKER\_TFS\_MAX\_ENQUEUED\_BATCHES | 进入队列排队的batch的最大数量 | 对于离线推理，这个值可以设置的很大，只要不OOM就可以；对于在线推理，为了减少排队导致的更大的延迟，把它设置为等于SAGEMAKER\_TFS\_NUM\_BATCH\_THREADS是一个不错的选择。 |

上面这些参数的设置可以参考[TFS的官网](https://github.com/tensorflow/serving/blob/r2.0/tensorflow_serving/batching/README.md)。**这些参数没有适合所有场景的最佳设置，一般都是根据业务场景，模型，推理设备，以及吞吐量和高分位数延迟需求来调试的，参考上表中的备注来进行尝试通常是一个好的起点**。

1. 与推理加速相关的配置

|  |  |  |
| --- | --- | --- |
| 环境变量 | 介绍 | 备注 |
| SAGEMAKER\_TFS\_INTRA\_OP\_PARALLELISM | 设置TF的单个运算符op 进行内部运算并行度的最大值 | 设置它为推理实例的VCPU的数量或者物理core的数量是一个不错的起点。 |
| SAGEMAKER\_TFS\_INTER\_OP\_PARALLELISM | 设置TF同时运行多个不同运算符op的最大值（它会共享执行op内部运算的线程池） | 设置它为推理实例的VCPU的数量或者物理core的数量是一个不错的起点。 |
| TF\_DISABLE\_MKL | 设置是否使用Intel的mkldnn库对神经网络在CPU设备上加速 | SageMaker内建的TFS的CPU容器缺省会安装并使用mkldnn做推理的加速；**有时候可能不用mkldnn会更快**。 |
| TF\_DISABLE\_POOL\_ALLOCATOR | mkldnn的内存池分配器 | 在disable mkldnn的时候，除了TF\_DISABLE\_MKL要设置为1，这个环境变量也要设置为1 |
| OMP\_NUM\_THREADS | mkldnn用于对运算符计算的线程池数量 | 设置它为推理实例的VCPU数量或者物理core的数量是一个不错的起点 |

当使用CPU设备进行推理时，如果要启用mkldnn，可以设置OMP\_NUM\_THREADS。不管是否禁用mkldnn，设置SAGEMAKER\_TFS\_INTRA\_OP\_PARALLELISM和SAGEMAKER\_TFS\_INTER\_OP\_PARALLELISM对推理加速都有帮助。

当使用GPU设备推理时，模型前向推理的运算都是在GPU上进行，因此设置SAGEMAKER\_TFS\_INTRA\_OP\_PARALLELISM和SAGEMAKER\_TFS\_INTER\_OP\_PARALLELISM这两个环境变量的意义不大。如果在保存模型的时候手动设置了serving\_input\_receiver\_fn 做数据的预处理，并且这个预处理的运算比较复杂，那么即使在GPU设备做推理的情况下，设置SAGEMAKER\_TFS\_INTRA\_OP\_PARALLELISM和SAGEMAKER\_TFS\_INTER\_OP\_PARALLELISM这两个环境变量对于TFS进行这个预处理也是很有帮助的，因为在数据的预处理中涉及到的TFS OP可以使用更多的CPU来加速。

1. 案例分享

背景：

借助SageMaker Inference Endpoint内建的TFS对推荐系统中的排序模型做serving。当前召回候选集的itemid数量是500个，单个itemid对应的样本/特征向量是4KB。

需求：

从推荐服务器侧对500个特征向量从发送开始到收到打分结果在50～100ms以内。

经过如下一系列的优化过程，可以满足该客户的需求（**从最开始的470ms减少到100ms以内**）：

1. **推理实例的机型选择**  
   这里使用的排序模型是一个小型的DNN模型，用GPU做推理的性价比不高 ，因此考虑用CPU实例做推理。

该客户最开始使用的机型是M5系列，对于机器学习推理这样的计算消耗型的应用，用CPU来推理的话，更建议使用C5系列这样的计算优化型机型。

为了更好的适应自动扩容，一般不建议使用很大的机型做推理；同时考虑到单个请求的payload大小以及实例的带宽，使用ml.c5.4xlarge是一个不错的尝试起点（这里需要注意请求发起侧的实例的带宽不要成为性能瓶颈）。

1. **启用 TFS server side batch**

根据上文提到的内建的TFS的server side batch相关的参数来进行尝试。

由于参数的组合情况比较多，这个尝试需要进行多次，可参考server side batch表格的备注列。

1. **TFS的op并行度和mkldnn的相关设置**

根据上文提到的TFS的inter op和intra op并行度以及mkldnn相关的环境变量来尝试。

在有些情况下，禁用mkldnn会有更快的推理速度（这个项目中正是disable mkldnn后推理速度更快）。一般建议还是对比禁用mkldnn与启用mkldnn并配置线程池的情况下，哪种推理速度更快。

1. **通信API的选择**

在SageMaker for TFS的推理容器中，缺省是使用REST API与TFS进程进行通信的。在推理容器内如果想使用gRPC API与TFS进行通信的话，需要借助inference.py来实现。

在实验时发现，当请求的payload比较小（比如200KB）的时候，推理容器内使用REST API的整个延迟要比使用gRPC API的小，当请求的payload比较大（比如4MB）的时候，gRPC API会比REST API快很多。

1. **序列化与反序列化的选择**

在使用gRPC API的时候，经过实验，对请求的payload使用JSON序列化与反序列化比使用string序列化和反序列化的性能差很多。

除了请求的payload的序列化与反序列化，响应的payload的序列化和反序列化也会影响性能。对于排序模型这个场景，把响应的numpy array作string序列化比把numpy array作json序列化或者作binary byte序列化要效率好。

1. **推荐服务器侧的大payload请求的拆分**

在推荐服务器侧把payload大的请求拆分为多个payload小的请求（每个拆分后的小请求用不同的线程来发送，不同的线程最好使用不同的sagemaker-runtime client对象。这种使用方式比拆分的所有小请求都使用同一个sagemaker-runtime client对象的延迟要小），并发发送给SageMaker Endpoint，之后在推荐服务器侧把多个小的请求对应的打分结果合并做最后的整体排序。

在客户的这个项目中，当推理容器使用gRPC API并且使用一个大payload的单个请求时，通过上面这些优化可以把总的延迟控制到150ms左右；当推理容器内使用REST API的方式（且不使用inference.py），把召回结果集对应的500个样本的大请求拆分为10个并发的小请求（每个小请求包含50个样本/特征向量），整个召回结果集的最终打分的总的延迟基本在100ms以内。

1. 示例代码：

下面先给出inference.py的参考代码片段：

import json

import numpy as np

import grpc

from tensorflow.compat.v1 import make\_tensor\_proto

from tensorflow\_serving.apis import predict\_pb2

from tensorflow\_serving.apis import prediction\_service\_pb2\_grpc

import os

import time

# default to use of GRPC

PREDICT\_USING\_GRPC = os.environ.get('PREDICT\_USING\_GRPC', 'true')

print('PREDICT\_USING\_GRPC')

print(PREDICT\_USING\_GRPC)

if PREDICT\_USING\_GRPC == 'true':

USE\_GRPC = True

else:

USE\_GRPC = False

MAX\_GRPC\_MESSAGE\_LENGTH = 512 \* 1024 \* 1024

options = [

('grpc.max\_send\_message\_length', MAX\_GRPC\_MESSAGE\_LENGTH),

('grpc.max\_receive\_message\_length', MAX\_GRPC\_MESSAGE\_LENGTH)

]

def handler(data, context):

if context.request\_content\_type == **'X-protobuf'**:

d = data.read()

channel = grpc.insecure\_channel(f'0.0.0.0:{context.grpc\_port}', options=options)

request = predict\_pb2.PredictRequest()

#**String Deserialization is faster than json Deserialization**

**request.ParseFromString**(d)

stub = prediction\_service\_pb2\_grpc.PredictionServiceStub(channel)

result\_future = stub.Predict.future(request, 5)

output\_tensor\_proto = result\_future.result().outputs[**'prob'**]

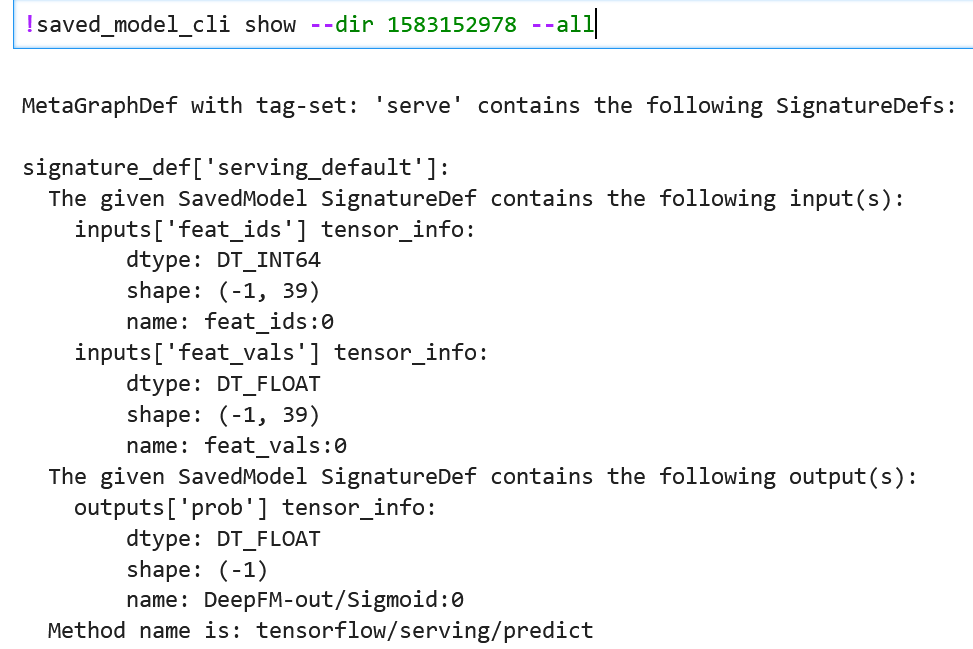
output\_shape = [dim.size for dim in output\_tensor\_proto.tensor\_shape.dim]

output\_np = np.array(output\_tensor\_proto.float\_val).reshape(output\_shape)

prediction\_json = json.dumps({'prob': output\_np.tolist()})

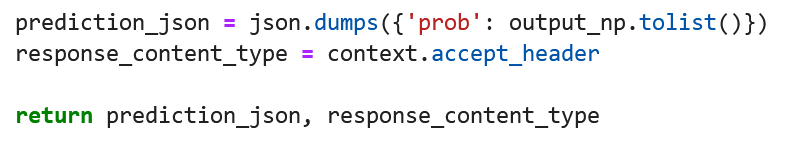
response\_content\_type = context.accept\_header

return prediction\_json, response\_content\_type

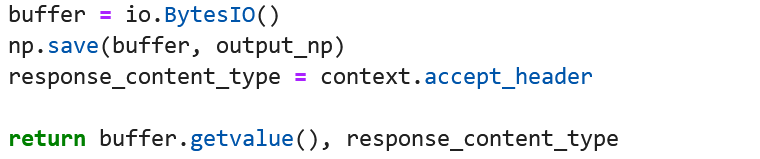
上面代码中对protocol buffer request使用了string反序列化（因为在客户端对请求进行了string序列化）。代码中出现的”prob” 需要修改为使用saved\_model\_cli命令行查看你的模型的某个output tensor的名字。我这里用的模型的output tensor的名字是”prob”，如下所示：  


上面代码使用的是对响应的numpy array序列化为json，也可以尝试把numpy array序列化为string或者binary byte，来看看哪个方式效率更高。这三种方式的区别反应在inference.py代码中就是最后几行不同：

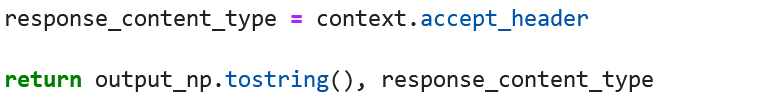
* 对于numpy array序列化为json，最后几行代码如下：



* 对于numpy array序列化为binary byte，最后几行代码如下：



* 对于numpy array序列化为string，最后几行代码如下：



下面是调用SageMaker Python SDK来部署模型并且简单测试在线推理的代码片段，供参考：

from sagemaker.tensorflow.serving import Model

import sagemaker

role = sagemaker.get\_execution\_role()

model = Model(

source\_dir = "code",

entry\_point='inference.py',

model\_data=**'s3://YOUR\_BUCKET/model.tar.gz'**, #change to your model s3 uri

role=role,

framework\_version="2.3",

env = {

'SAGEMAKER\_TFS\_ENABLE\_BATCHING': 'true',

'SAGEMAKER\_TFS\_BATCH\_TIMEOUT\_MICROS': '1000',

'SAGEMAKER\_TFS\_MAX\_BATCH\_SIZE': '128000',

'SAGEMAKER\_TFS\_NUM\_BATCH\_THREADS':'16',

'SAGEMAKER\_TFS\_MAX\_ENQUEUED\_BATCHES':'16',

'SAGEMAKER\_TFS\_INTER\_OP\_PARALLELISM':'16',

'SAGEMAKER\_TFS\_INTRA\_OP\_PARALLELISM':'16',

'TF\_DISABLE\_MKL':'1',

'TF\_DISABLE\_POOL\_ALLOCATOR':'1',

"SAGEMAKER\_GUNICORN\_WORKERS":"8"

}

)

predictor = model.deploy(initial\_instance\_count=1, instance\_type='ml.c5.4xlarge')

除了提供inference.py代码，你还需要提供一个依赖包的requirements.txt文件，依赖的安装包是tensorflow-serving-api和numpy。关于上面代码中的环境变量的设置具体参考前文相关章节（上面的环境变量设置disable 了mkldnn，对于客户的模型来说，enable mkldnn并且设置mkldnn的线程池，推理速度比disable mkldnn的情况差很多）。

下面的代码用来构造protocol buffer请求（这里为了方便模拟payload比较大的请求，直接复制一个样本为500个样本；并且以 DeepFM模型为例），并把请求作string序列化。

from tensorflow\_serving.apis import predict\_pb2

from tensorflow\_serving.apis import prediction\_service\_pb2\_grpc

import numpy as np

from tensorflow.compat.v1 import make\_tensor\_proto

import time

t = [1,2,3,4,5,6,7,8,9,10,11,12,13,15,555,1078,17797,26190,26341,28570,35361,35613,35984,48424,51364,64053,65964,66206,71628,84088,84119,86889,88280,88283,100288,100300,102447,109932,111823]

f = [0.05,0.006633,0.05,0,0.021594,0.008,0.15,0.04,0.362,0.1,0.2,0,0.04,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1]

inputs = {}

inputs["feat\_ids"] = [t for i in range(0,500)]

inputs["feat\_vals"] = [f for i in range(0,500)]

features = ['feat\_ids', 'feat\_vals']

options = [

('grpc.max\_send\_message\_length', 1024\*1024\*1024),

('grpc.max\_receive\_message\_length', 1024\*1024\*1024)

]

request = predict\_pb2.PredictRequest()

request.model\_spec.name = 'model'

request.model\_spec.signature\_name = **'serving\_default'**

request.inputs["feat\_ids"].CopyFrom(make\_tensor\_proto(np.array(inputs["feat\_ids"], dtype="int64")))

request.inputs["feat\_vals"].CopyFrom(make\_tensor\_proto(np.array(inputs["feat\_vals"], dtype="float32")))

payload = request.**SerializeToString**()

您可以按照上面的代码根据相应模型的input tensor来进行构造，上面使用的signature\_name对应您在保存模型时设置的signature，如果没有设置，缺省是'serving\_default'。

接下来是三种不同的response的序列化方法对应的客户端反序列化的代码。

1. response的numpy array序列化为JSON的情况，在客户端反序列化的时候使用json.load()的代码片段如下：

import boto3

import time

import json

client = boto3.client('sagemaker-runtime')

endpoint\_name = **'your\_endpoint\_1'**

content\_type = "**X-protobuf**"

list\_1 = []

for i in range(0,500):

time\_start = time.time()

response = client.invoke\_endpoint(

EndpointName=endpoint\_name,

ContentType=content\_type,

Body = payload

)

result = **json.loads**(response['Body'].read().decode())[**'prob'**]

time\_end = time.time()

list\_1.append(time\_end-time\_start)

print(sum(list\_1)/len(list\_1))

因为这里我们会使用inference.py做数据的前处理和后处理，所以上面代码中的content\_type可以随便写，只要和inference.py代码中期望的content\_type保持一致即可。

1. response的numpy array序列化为binary byte的情况，在客户端反序列化的时候使用np.load(io.BytesIO())的代码片段如下：

import io

list\_1 = []

for i in range(0,500):

time\_start = time.time()

response = client.invoke\_endpoint(

EndpointName=**'your\_endpoint\_2'**,

ContentType=content\_type,

Body = payload

)

result = **np.load(io.BytesIO**(response['Body'].read()), allow\_pickle=True)

#print(result)

time\_end = time.time()

list\_1.append(time\_end-time\_start)

print(sum(list\_1)/len(list\_1))

1. response的numpy array序列化为string的情况，在客户端反序列化的时候使用np.frombuffer()的代码如下：

import io

list\_1 = []

for i in range(0,500):

time\_start = time.time()

response = client.invoke\_endpoint(

EndpointName=**'your\_endpoint\_3'**,

ContentType=content\_type,

Body = payload

)

result = **np.frombuffer**(response['Body'].read())

#print(result)

time\_end = time.time()

list\_1.append(time\_end-time\_start)

print(sum(list\_1)/len(list\_1))

1. 总结

本文详细介绍了在SageMaker中的内建TFS推理容器的一些细节，并通过一个真实的项目讲解了加速推理的过程和思路，以及相关的参考代码，感谢大家的阅读。