Tensorflow Serving基础与应用

Ruichen Wang

July 12, 2019

Abstract

Tensorflow serving 基础与应用，介绍tensorflow serving基本概念，应用示例和性能评估。

1 介绍

TF Serving是TensorFlow官方提供的一套用于模型服务的框架，目标就是实现在线，低延迟的预测服务。它的突出优点是:和TensorFlow无缝链接，可以将训练好的机器学习模型部署到线上，支持多模型多版本同时serving，支持HTTP/gRPC作为接口接受外部调用。TensorFlow Serving支持模型热更新与自动模型版本管理,对扩展性的支持和性能都非常好。目前业内有很多公司已经采用此技术来提供服务，也是阿里云提供的深度学习预测服务的唯一选择。

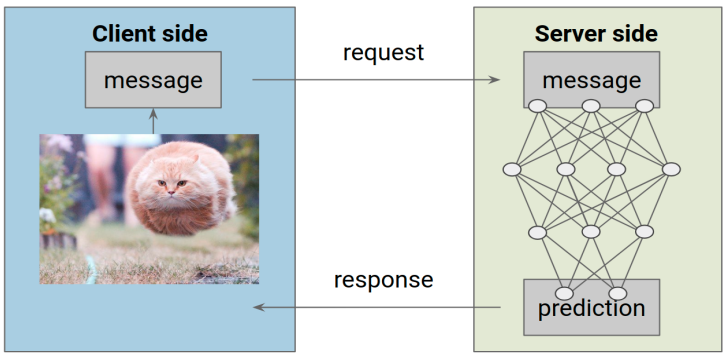

Figure 1: Tensorflow Serving

features:

- Can serve multiple models, or multiple versions of the same model simultaneously

- Exposes both gRPC as well as HTTP inference endpoints
- Allows deployment of new model versions without changing any client code
- Supports canarying new versions and A/B testing experimental models
- Adds minimal latency to inference time due to efficient, low-overhead implementation
- Features a scheduler that groups individual inference requests into batches for joint execution on GPU, with configurable latency controls
- Supports many servables: Tensorflow models, embeddings, vocabularies, feature transformations and even non-Tensorflow-based machine learning models

在实际场景中，算法的整个流程应该包括数据准备和预处理，模型训练，预测服务三大块。Tensorflow Serving主要应用于模型的预测服务中。针对我们的信息流推荐实际业务，采用deepfm模型，完全可以实现毫秒级的响应。批量预测1000个case也只需要0.3ms左右的时间。

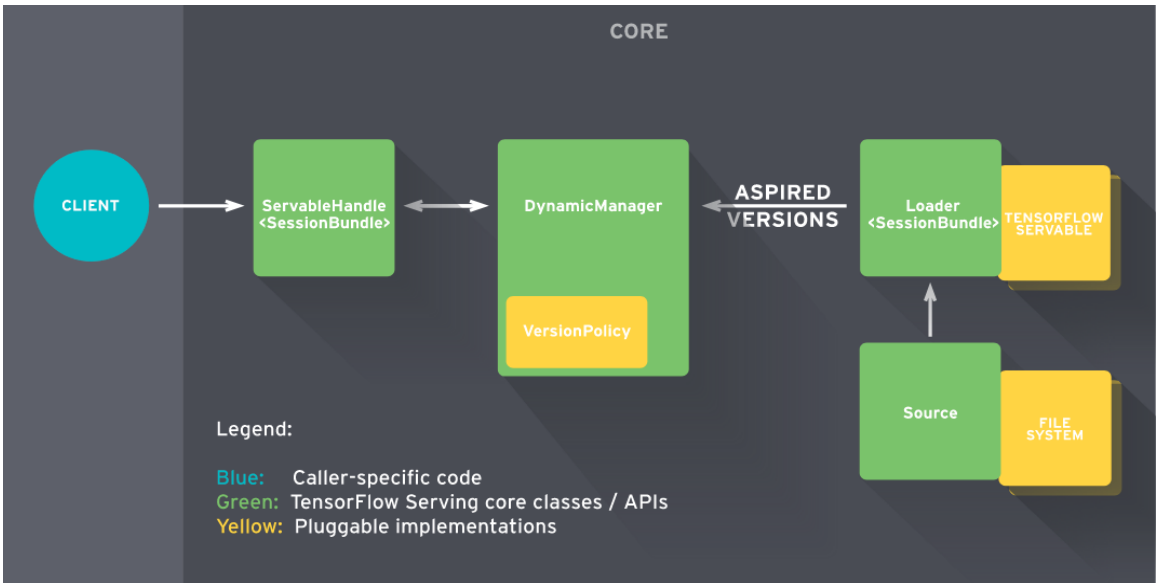

Figure 2: life of a servable

Source组件为特定的模型版本创建一个Loader，其中包含用作serving的所有元数据。同时通知Manager最新的版本号。由manager来决定是否需要重新加载之前版本或者最新版本。客户端请求时可以默认使用最新版本，也可以指定访问某一固定版本。

一些Extension组件介绍:

- Version Policy - 主要提供两种版本控制协议Availability Preserving Policy 和Resource Preserving Policy
- Source
- Loader
- Batcher - 用于控制batch size, 线程数，batch队列，超时时间等

2 生成模型

Tensorflow提供了多种保存模型的方法。

```
tf.saved_model.simple_save (keras)
estimator.export.build_parsing_serving_input_receiver_fn
estimator.export.build_raw_serving_input_receiver_fn
```

最简单的可以使用simple save方法，以key-value的格式，指定模型输入和输出。一般结合keras model一起使用。这也是tensorflow 2.0以后比较的方法。另外使用estimator.export提供的两个接口，更方便结合feature column导出。在目前tensorflow1.14适用性更广一些。

```
test_serving
└──mnist_model_for_serving
|   └──1
|       |   variables
|       |   saved_model.pb
```

Figure 3: saved model path

```
(base) wangrc@wangrc:~/mnist_model_for_serving/1$ saved_model_cli show --dir . --all

MetaGraphDef with tag-set: 'serve' contains the following SignatureDefs:

signature_def['serving_default']:
  The given SavedModel SignatureDef contains the following input(s):
    inputs['input_image'] tensor_info:
        dtype: DT_FLOAT
        shape: (-1, 28, 28, 1)
        name: input_1:0
  The given SavedModel SignatureDef contains the following output(s):
    outputs['dense_1/Softmax:0'] tensor_info:
        dtype: DT_FLOAT
        shape: (-1, 10)
        name: dense_1/Softmax:0
  Method name is: tensorflow/serving/predict
```

Figure 4: savedmodelcli

模型文件保存后，会生成对应版本号的文件夹，在同一个模型目录下，tesorflow serving会默认自动加载版本号最新的模型文件，tensorflow提供了bash命令来查看保存的模型接口和输出结果，方便client调用。

保存模型的时候，可以定义Classify, Regress或者Predict API。以predict为例，在保存模型时指定predict的signature key, 调用时只要在客户端请求指定的method name接口就可以了。

- Regress : 1 input tensor, 1 output tensor.
- Classify : 1 input tensor, output classes & scores.
- Predict : arbitray many input and output tensors.

```
export_outputs = {
    tf.saved_model.signature_constants.DEFAULT_SERVING_SIGNATURE_DEF_KEY: estimator.export.PredictOutput(
        predictions)}

if mode == estimator.ModeKeys.PREDICT:
    return estimator.EstimatorSpec(
        mode=mode,
        predictions=predictions,
        export_outputs=export_outputs)
```

Figure 5: predict api

3 提供服务

有了模型文件，我们就可以部署我们的模型服务了。主要有以下三种方案:

- tensorflow model server + 单节点
- docker + 单节点
- docker + k8s + 集群部署

在测试阶段，可以采用第一种或者第二种，可以方便调试与测试模型性能等指标。如果真正部署至生产环境，就需要依托k8s的的平台，提供服务注册与发现，负载均衡，链路监控，网关路由等一系列微服务功能。

- 如果是ubuntu系统，可以方便的使用apt安装tensorlfow model server。其他情况，推荐采用docker方法。

参考链接`https://www.tensorflow.org/tfx/serving/setup`。

- 使用docker比较简单，只需要拉取tensorlfow serving镜像，然后docker run一下就行了。推荐直接采用docker的方法来测试模型服务。这也是最方便的使用gpu版本的serving的方法。在官方docker镜像中，默认使用8500作为grpc调用接口，8501作为REST调用接口。可以通过-p命令映射到宿主其他端口上。

```
docker pull tensorflow/serving
```

docker 启动服务：

```
# For gRpc，默认端口8500
docker run -p 8500:8500 --mount type=bind,source=/home/wangrc/test_serving/mnist_model_for_serving,target=/models/mnist -e MODEL_NAME=mnist -t

# For REST, 默认端口8501
docker run -p 8501:8501 --mount type=bind,source=/home/wangrc/test_serving/mnist_model_for_serving,target=/models/mnist -e MODEL_NAME=mnist -t
```

Figure 6: docker service start

- 在生产环境，模型服务一般需要在集群上部署，需要支持较大的QPS。可以采用k8s来编排部署。可以自己指定副本数量与计算资源。在副本出现问题的时候k8s会重新拉起，同时保证分配计算资源。

```yaml
namespace: algorithms

deepfm:
  componentName: deepfm
  enabled: true
  replicas: 2
  image:
    repository: harbor.2345.cn/algorithms/deepfm
    tag: latest
    imagePullPolicy: Always
  resources:
    limits:
      cpu: 4000m
      memory: 4Gi
    requests:
      cpu: 100m
      memory: 128Mi
  nodeSelector: {}
  affinity: {}
  tolerations: []

service:
  deepfm:
    type: NodePort
    gRPCPort: 8500
    httpPort: 8501
  demo:
    type: ClusterIP
    port: 8501

ingress:
  deepfm:
    host: algorithmsdeepfm.2345.cn
```

Figure 7: k8s yaml

4 客户端调用

在目前的tensorflow提供了查看模型服务状态的方法，同时也为客户端提供了gPRC和REST两种调用模式。

REST 使用REST方法时，数据使用json字符串格式传输，使用utf-8编码。值得注意的是，如果输入的数据是二进制格式的，需要将字符串转化成Base64的格式。另外，一般我们常用的float类型的特征经常会出现inf或者nan。默认的json也不能识别此类数据。需要使用json parser进行转换。

```
{
  // (Optional) Serving signature to use.
  // If unspecifed default serving signature is used.
  "signature_name": <string>,

  // Input Tensors in row ("instances") or columnar ("inputs") format.
  // A request can have either of them but NOT both.
  "instances": <value>|<(nested)list>|<list-of-objects>
  "inputs": <value>|<(nested)list>|<object>
}
```

Figure 8: request format

```
import json
data = json.dumps({"signature_name": "serving_default", "instances": [test_images[rando].tolist()]})
# print('Data: {} ... {}'.format(data[:50], data[len(data)-52:]))

import requests
headers = {"content-type": "application/json"}
json_response = requests.post('http://localhost:8502/v1/models/fashion_model/versions/1:predict', data=data, headers=headers)
print(json_response.text)
predictions = json.loads(json_response.text)['predictions']
```

Figure 9: rest request demo

gPRC 在实际应用中，如果特征维度比较高，或者是直接传输图像数据，这时候数据量一般比较大，进行json转化会比较耗时，拼接成的字符串也会较大，传输时间往往会是瓶颈。这时候就可以考虑使用grpc调用。一般情况下，同样的数据，使用grpc实际传输字节只有json的一半，配合serialize example格式，模型预测速度也有3倍左右的提高。

在用docker模型serving的时候，可以同时对外映射模型的grpc和rest接口，客户端可以根据应用需要选择对应的接口。

性能比较：

- REST (serialized once)
  - Inference rate: 7,680 img/sec
  - Network: 620 MB
- gPRC (serialized once)
  - Inference rate: 25,961 img/sec
  - Network: 320 MB

Figure 10: grpc deepfm speed test

```
for i in range(1000):
    feature_dict = {'SepalLength': _float_feature(value=np.random.random()),
                    'SepalWidth': _float_feature(value=np.random.random()),
                    'PetalLength': _float_feature(value=np.random.random()),
                    'PetalWidth': _float_feature(value=np.random.random())}

    example = tf.train.Example(features=tf.train.Features(feature=feature_dict))
    serialized = example.SerializeToString()
    batching.append(serialized)

request.inputs['examples'].CopyFrom(
    tf.make_tensor_proto(batching, shape=[len(batching)]))

start = time()
result_future = stub.Predict.future(request, 5.0)
elapsed = (time() - start)
prediction = result_future.result().outputs['probabilities']
print(prediction)
print("Time used:{0}ms".format(round(elapsed * 1000, 2)))
```

Figure 11: grpc client demo

5 版本切换warmup

tf serving在模型版本切换的时候，初始的几个case响应时间会明显高于正常水平。这种情况主要是因为tensorflow 运行时采用lazily initialization的方式。tensorflow可以自定义warm up request。保存一个用于warmup的tfrecords。这样每次模型加载的时候，会先调用tfrecords里的样本先做几次预测，初始化相关组件。

6 demo相关代码

ha05/home/wangrc/demo/

7 答疑