模型推理是将机器学习模型应用到业务数据并提供对该数据的洞察，其经常在业务系统中作为一个模块提供服务，作为整个机器学习生命周期中的一个必不可少的阶段，它的重要性毋庸置疑。不同的ML任务和业务应用场景下对于模型推理的诉求也是不同的，Amazon SageMaker在模型推理方面提供了众多的功能来满足不同场景下的需求。对于任何一种推理的场景，机器学习工程师都希望能尽量提升推理服务器侧的吞吐以及降低推理的整个延迟，接下来我们从工程角度讨论一下机器学习推理这个话题**。**

1. **多维度的ML推理的诉求**
2. **离线推理和在线推理**

简单来讲，离线推理和在线推理有如下的区别：

* 在线推理的实现复杂度要比离线推理高；
* 在线推理的金钱成本比离线推理高；
* 离线推理对延迟一般没有很严格的要求。

需要注意的是，并不是所有的业务场景都需要在线推理。

|  |  |
| --- | --- |
| **Domain** | **离线推理场景的例子** |
| 时间序列预测 | 比如某产品的销量预测任务 |
| 传统的表格数据预测类任务 | 比如用户付费预测任务，用户流失预测任务 |
| 计算机视觉 | 比如电商商品图片分类任务 |
| 自然语言处理 | 比如用户情感分析任务 |
| 语音处理 | 比如客服系统的用户语音留言转录任务 |
| 异常检测 | 比如定期对某种材料的缺陷检测任务 |
| 搜索，推荐系统，计算广告 | 比如电视节目的离线推荐 |

对于时间序列预测和传统的表格数据预测类任务，他们的大部分任务用离线推理；对于搜索，推荐系统和计算广告这三大领域，他们的大部分任务需要在线推理，有些任务可以用离线推理；而对于计算机视觉，自然语言处理，语音处理和异常检测这些领域，他们的任务是离线推理还是在线推理更多是取决于具体的业务需求。

**在某些情况下，可以把在线推理降级为离线推理**（来自Netflix推荐系统的实践），比如在线推理服务发生故障且短时间不能恢复时，可以用预先保存的离线推理的结果来作过渡。

1. **同步推理和异步推理**

简单来讲，同步推理和异步推理的语义上的区别是：发起推理请求的一方得到的响应是否是推理的结果。一般来说，异步推理会使整个数据处理流程变得复杂（进而导致架构也变得复杂），**因此在决定采用异步推理之前，要想清楚同步推理是否能满足你的业务需求**。

常见的使用异步推理的场景是：

* **缓解上游的业务服务器和下游的推理服务器的处理速度的严重不匹配**。
* **避免推理时间太长导致对请求发起侧的长时间阻塞。**
* **绕过推理服务侧对最大请求payload size的限制**。

Amazon SageMaker提供的异步推理功能的架构如下： 
                异步推理的架构图，显示用户如何调用终端节点以及终端节点的响应方式。
            

1. **延迟敏感**

延迟敏感指的是从终端用户侧，或者从当前处理节点，或者从推理请求发起侧来说，需要很严格的分位数延迟（很多客户不只是看平均延迟以及中位数延迟，还关心P90/P99的分位数延迟）。

下面给出几个对延迟比较敏感的业务/领域：

* **DSP竞价广告场景，可接受的延迟经常是10ms以内**。

DSP的上游经常是ADX广告网络联盟，ADX的竞价最大延迟要求是100ms（即从ADX把竞价请求从自己那侧发出到收到DSP竞价结果的整个延迟要求在100ms内）。DSP把它与ADX之间通信的网络往返带入的抖动也会考虑进去（比如通常情况下，DSP与上游的ADX之间的网络往返延迟是20ms，但是偶尔在网络比较差的情况下这个网络往返延迟可能会达到60ms，抖动就是40ms），因此DSP希望自己从收到竞价请求到发送完竞价结果的整个延迟控制在10ms内，这样即使在网络抖动比较差的情况下，也不会影响DSP有效参与到ADX的广告竞价中。

* **推荐系统场景，可接受的延迟一般是100ms～500ms**。

这里的延迟一般指的是推荐服务器从接受到上游的曝光请求到把推荐结果发送回去的整个延迟。在实际的推荐系统的项目中，也经常会遇到客户对于排序任务的延迟要求是100ms级别，即把特征拼接好的召回结果发送出去到收到排序模型的打分结果这个过程的延迟（不包括召回结果集的特征拼接开销）**。**

* 除了搜推广领域，同声机器翻译，聊天窗口或者评论发布的实时敏感词过滤，自动驾驶中的图像语义分割等这些场景对延迟也比较敏感。

推理架构的拓扑常见有两种方式：

* **All in one process即单体服务**：

对于延迟最敏感的业务比如对接ADX的DSP竞价广告就经常使用这个拓扑（在实际的对接ADX的DSP项目中见到的也都是这样的拓扑）。

* **推理服务从整个业务逻辑中解耦出来作为单独的服务提供**：

从软件设计哲学来看，这个拓扑是最佳的（因为它降低了软件复杂度，更便于推理模块的迭代更新，更便于autoscaling，更便于维护管理），除非它无法满足延迟要求。

1. **金钱成本**

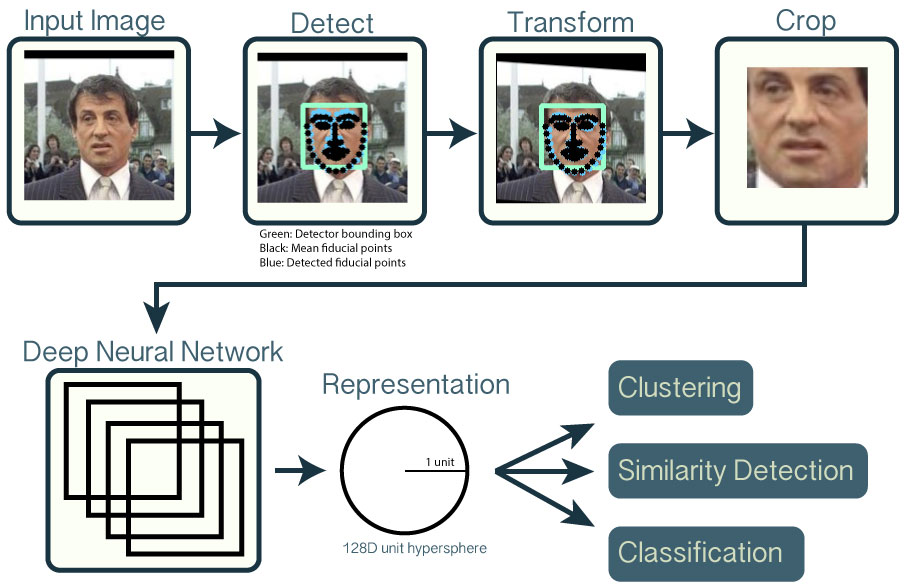
金钱成本在任何时候任何企业都会考虑的，我们可以从多个视角来看如何节省金钱成本。

* 视角一：从推理的加速方法和推理服务侧的QPS优化方法入手（对于这两种方式，我们后面会专门讲解）
* 视角二：对于在线推理的场景，SageMaker提供了如下几种方案来节约金钱成本（下面的三个方案今后都会有专题来介绍，这里只需要简单了解一下）：

|  |  |
| --- | --- |
| 方案 | 介绍 |
| SageMaker Serverless Inference Endpoint | **金钱成本节省原因**：只有来了请求才会使用计算资源；  **适用场景：**并发请求不大，并且能容忍cold start导致的延迟的场景（当前只是适用于CPU推理的场景） |
| SageMaker Multi-model Endpoint（MME） | **金钱成本节省原因**：多个模型共享一个推理实例的同一个serving container**；**  **适用场景：**推理的QPS不高且对推理延迟有一定容忍的场景 |
| SageMaker Multiple Container Endpoint（MCE direct模式） | **金钱成本节省原因**：在一个推理实例上，用多个container来serving不同框架且没有依赖的多个模型**；**  **适用场景：**推理的QPS不高且对推理延迟有一定容忍的场景 |

1. **模型之间的依赖**

在有些比较复杂的ML任务中，会涉及到多个模型，比如人脸识别的任务如下面所示（图片引用自[这里](https://cmusatyalab.github.io/openface/)）：



它涉及到如下的步骤（下面的#A，#B，#C三个步骤会涉及到三个模型）：

1. 先用人脸检测算法来检测是否有人脸并定位人脸；
2. 接着进行人脸对齐（一般使用人脸关键点检测来实现，把人脸转换为标准脸）；
3. 接着对标准脸进行人脸特征向量提取；
4. 最后把人脸特征向量与资料库中保存的可信的人脸特征向量进行相似度对比，找到相似度分数最高且超过阈值的对应的那个人。

模型之间的依赖关系的强弱可以细分为：

* **模型之间直接前后强依赖**

上游模型推理后的输出就是下游模型推理需要的输入（对于这种情况，SageMaker inference pipeline是一个不错的选择）。 比如OCR的二阶段pipeline实现方式：文字检测（找到文字bbox并抠图）---->文字识别（把抠图中的文字找出并输出为字符串）

* **处理链路上模型之间有先后的处理顺序**

对于这样的情况，有两种思路：

1. 模型推理单独放在SageMaker中，其他的逻辑放在SageMaker外面；
2. 所有的逻辑都放入SageMaker中，用inference pipeline实现。

理论上可以把复杂的处理过程切分为多个小的阶段，**根据下一个阶段需要的输入来构造当前阶段的输出**。在这样的方式下，就可以使用SageMaker 的inference pipeline。但是不建议这样做（尤其是使用SageMaker Endpoint做线上推理的时候），因为inference pipeline的本质上还是希望这个pipeline中的每个组件都是模型推理部分或者模型的特征工程处理/数据预处理/数据后处理。因此对于这样的情况，建议还是使用方案#A，也就是用一个上游的组件/服务来处理/编排这些逻辑，把涉及到模型推理相关的放在SageMaker 里面。

* **模型之间是层次依赖关系**：

模型之间是层次的关系。比如对于图片的层次分类，有一种方案就是作为pipeline来建模：第一个层级用单个模型来预测图片属于哪个一级类别；第二个层级是多个模型，每个一级类别用一个模型，预测图片属于哪个二级类别；依次类推直到第N个层级（用层次分类建模而没有选择把所有最细类别单独一个flatten模型建模，原因是用层次建模的话，模型学习起来的难度要相对低一些，但是会有误差传递问题）。

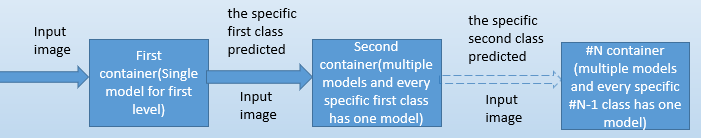
针对这种情况，如果是线上推理，可选择的方案是：

1. Inference pipeline方案：第一个容器放第一层级的单模型，第二个容器放第二个层级的多个模型，以此类推。

对于CPU推理实例，建议要让推理实例的内存足够大，可以容纳所有这些模型。

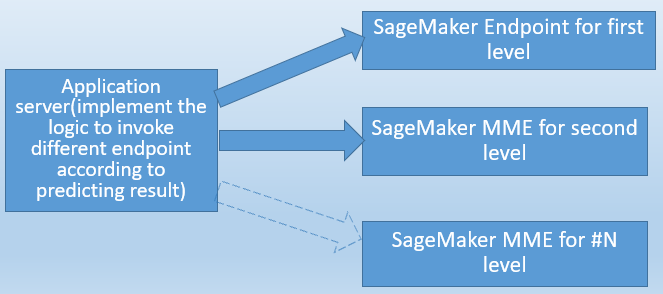
对于GPU推理实例，如果GPU实例只有1个GPU卡，那么GPU将是独占的，请求在GPU上处理时变成串行的（这里指的是不用vGPU或者Nvidia MPS或者手动CUDA高并发编程的情况）；如果想并发的处理多个inference pipeline请求，且每个容器中的所有模型需要的总的显存小于单个GPU卡的显存，并且容器数量小于单机GPU卡的数量，可以使用多卡的机器并且给每个容器绑定上不同的GPU卡。

拓扑参考如下：

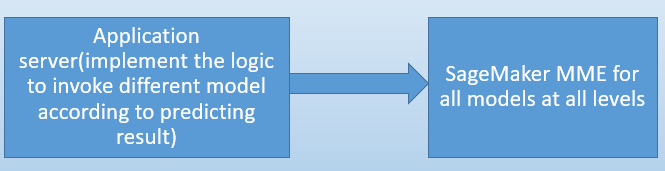


1. Multi-model Endpoint（MME）方案：第一个层级的单模型放在SageMaker Endpoint，第二个层级的多模型放在MME，第三个层级的多模型放在另一个MME，以此类推；或者把这些不同层级的所有模型都放在一个MME里面。

拓扑参考如下：



或者：



1. All in one process：把所有不同层级模型放在一个进程中，自己写逻辑来做推理。

**对于模型之间是层次依赖关系这样的场景，如果是延迟不敏感的业务并且推理实例的内存或者GPU显存够用，建议使用上面提到的inference pipeline方案；如果延迟不敏感但是推理实例的内存或者GPU显存不够用，那么可以考虑上面提到的MME方案。**如果是离线推理，inference pipeline和All in one process的方案都可以选择（细节同上），更建议使用inference pipeline。

1. **推理服务器侧的QPS优化**

只要是服务器侧，一定会关心吞吐量，推理服务器侧也不例外。而QPS是吞吐量的一个重要指标（吞吐量的另一个重要指标是并发数），也是我们这里主要讨论的指标。**单位时间内可处理完的请求数就越多，那么单个请求处理折算后的金钱成本就越低**。

|  |  |
| --- | --- |
| 方法 | 介绍 |
| Client side batch | 含义：客户端发送的单个推理请求中包括多个样本  使用场景举例：搜推广领域的排序模型的在线推理；支持batch的模型的离线推理（绝大部分模型都支持）  使用SageMaker的batch transform做离线推理时，使用multiple record策略的推理效率要比single record策略高很多。 |
| Server side batch | 含义：server侧把来自多个不同客户端请求中的少量样本组装为一个大的batch做一次模型的推理（而不是每个请求做一次推理）。  **Server side batch可以与client side batch结合使用**。  深度学习框架对server side batch的支持情况：Native的Tensorflow serving，MxNet serving，Pytorch serving都支持server side batch。 |
| 线程池/进程池 | 目的：为了能在server侧更好的做并行，充分的利用计算资源，可以使用线程池或者进程池方法。  使用场景举例：比如Sklearn serving经常会用Gunicorn + flask/falcon框架在CPU实例上来做推理，这个时候可以设置Gunicorn的worker/进程数量和/或者Gunicorn的线程数量。 |
| 请求发起侧的带宽限制和接受侧的带宽限制 | 目的：尽量不要让机器的带宽成为限制。  在进行压测，或者单个请求有很大的payload的时候，或者并发多个请求的时候，需要估算是否到达了发起侧和接受侧的带宽限制。 |

1. **模型推理的加速**  
   模型推理的加速本身是一个优化问题，我们需要找到影响速度的bottleneck，我们需要知道对于一个业务系统做推理优化涉及到的每个影响因素。我们通过对一个请求的整个处理过程的拆分提取出如下的这些重要的影响因素（从high level到low level）：

* 架构拓扑
* 服务之间的网络延迟
* 服务本身的处理效率
* 通信API的效率
* 序列化与反序列化的效率
* 模型本身的优化
* 推理设备的选择
* 推理框架本身的优化

我们来对每个影响因素进行详细解读。

1. **架构拓扑**

架构的不同拓扑决定了一个请求的处理路径的长度。一般来说，请求的处理链路越长，延迟也会越高。因此这里需要权衡，是选择使用微服务的设计思路还是需要追求最低的延迟。大部分场景下，我们都是建议在延迟可以接受的情况下用微服务这样的设计。

场景举例:

* DSP竞价广告业务, 几乎都是all in one process的方式。

DSP竞价广告的工作性质决定了它对整个端到端延迟（一般要求10ms内）要求非常严格。

* 其他大部分场景, 模型推理服务可以考虑从业务服务器逻辑中单独剥离出来。

比如使用SageMaker Endpoint for built-in TFServing做推荐系统的排序模型的线上推理的话，一个建议的架构拓扑是：

Recommender server（召回逻辑可以在Recommender server中实现，也可以单独剥离出去作为一个服务）---> SageMaker Endpoint ---> Nginx ---> guincorn(optional) --->TFS

1. **服务之间的网络延迟**

网络延迟也是经常需要考虑的点，尤其是请求发起端和推理服务端不在一个区域或者是两个不同的云服务提供商的情况。如果请求发起端和推理服务端都在同一个AWS的区域, 那么他们之间的网络往返延迟是10ms以内。因此，整个业务系统的每个组件的置放以及这个置放导致的网络延迟是否在可接受的范围，甚至网络加速方案都需要考虑进来。

* 比如有的客户由于种种原因就会有业务系统的上游组件在云厂商A，下游组件在云厂商B。这个时候两个云厂商之间的通信要走公网，那这个延迟就是不可控的，这个时候用Amazon Direct Connect专线是一个可选的加速方案。

1. **服务本身的处理效率**

在整个请求的处理过程中会涉及到除了模型推理服务外的其他服务，不管是什么性质的服务，**服务本身的处理以及代码编写是否高效都是非常重要的**。另外，对于推理服务来说，它是一个计算密集型的任务，因此**把一些复杂的预处理的逻辑放在它的上游客户端更好**（有点像瘦服务端的意思）。

* 如果还在模型实验阶段，并且对数据预处理逻辑的改动比较频繁，这个时候为了尽量不让上游客户端感知，把复杂的预处理逻辑放在推理服务端侧做过渡也可以接受，但是当模型稳定以后，也就是模型持续在线上稳定运行且不需要修改预处理逻辑以后，可以考虑把这个预处理逻辑放在上游客户端。

真实的例子：

* 比如Redis可以作为online feature store来提供特征的获取, 那么Redis集群的性能以及Redis客户端的使用方式就很重要。之前遇到过一个场景，要获取500个itemid的特征， 相比使用Redis客户端发出了500个请求，更好的方式是使用Redis的pipeline功能在一个请求中发送多个命令。
* 比如想拆分一个payload很大的请求为多个小的请求，使用goroutine同时发送多个请求给推理服务侧。为了更好的并行，这里设置goroutine可使用的CPU数量为当前机器的总的CPU数量或者物理core的数量是一个更好的选择。
* 比如被频繁调用的某个函数中会使用循环体来对一个变量进行相对复杂的操作，而该变量本质上是一个常量（比如feature字典的初始化或者Vocabulary字典的加载）。那最好的处理方式是把这个循环体提取到函数外面，且把该变量作为一个全局变量执行一次初始化。
* 比如发送一个JPEG图片到模型推理侧进行JPEG解码并推理，比在发送侧对JPEG图片解码后再发送给模型推理侧进行推理更好（JPEG解码后的payload要远大于JPEG图片本身的payload，从而带来更多的网络发送延迟和接受延迟）。

1. **通信API的效率**

**不同的通信API的效率有很大的差别**。对于推理服务的两种常见的API是gRPC API和REST API，这两种API的简单对比如下：

* gRPC基于HTTP 2，REST API基于HTTP1.1；
* gPRC默认使用Protocol buffers来序列化负载数据，REST API主要依赖JSON或者XML来收发数据。
* 相对于REST API代码的简单易用，使用gPRC的代码要复杂一些，可读性差一些。

**是不是任何情况下，gRPC的性能都比REST要高**？

* 通过实际项目发现，当请求的payload比较小（比如200KB）， REST API通常会快一点；当请求的payload比较大（比如4MB），gRPC API会比REST API快很多。

1. **序列化与反序列化的效率**

数据在进行网络传输时，需要在发送侧把内存中的对象序列化为适合在网络上传输的字节流，对应在服务侧需要把从网络上接受到的字节流反序列化为内存中的对象以方便程序操作。不同的序列化和反序列化的方法的效率也不一样：比如对Ndarray做 JSON序列化与反序列化比对Ndarray做String序列化和反序列化的性能差。

1. **模型本身的优化**

这里从模型层面入手考虑如何能降低纯粹的模型推理时间，有下面三种思路可以尝试：

|  |  |
| --- | --- |
| 思路 | 介绍 |
| 选用小的模型建模ML任务 | 在模型精度/效果可以接受情况下，使用小模型（模型参数少）  尤其在端侧做模型部署的时候，这个方法比较适用。 |
| 模型编译 | 模型编译只是为了加速推理，但是编译后的模型本身的大小不一定比编译前的模型小。但是它的推理runtime占用的总的内存资源一般要小于原生框架做推理时占用的内存资源。  常见的模型编译工具：TensorRT，TVM， Tensorflow lite |
| 模型压缩 | 模型压缩的目的很直接，就是在模型精度损失很小的情况下通过某种方法把模型的大小变小。  常见的模型压缩的方法：量化，剪枝，知识蒸馏。 |

除了模型压缩方法中可以做量化，常见的模型编译工具比如TensorRT和 TVM也会做模型参数的精度量化。从上面的表中可以看出，模型编译和模型压缩追求的都是加速模型推理，尽管使用方法非常不同。相对于模型编译，模型压缩使用起来要更复杂。有些情况下，两者可以结合使用，比如先把大模型用知识蒸馏变成小模型，然后在使用TVM/TensorRT进一步优化。在SageMaker中，通过使用SageMaker Neo功能可以简单的通过设置参数就可以对某些模型比如图像分类模型和XGBoost模型针对目标硬件上做编译，而不需要关心具体细节。

一般来说，如果模型运行在server侧，并且在纯的模型推理时延可以接受的情况下，模型压缩或者模型编译用的不多，但是如果模型运行在端侧（资源受限）的话，模型压缩或者模型编译几乎是一定需要的。

1. **推理设备的选择**

用作模型推理的设备有很多种，在亚马逊云科技上的可选的设备有CPU，GPU，自研机器学习推理专用芯片Inferentia，EIA（EIA支持直接附加到SageMaker Endpoint的CPU实例上），而不同的设备还有很多不同的细分类型。

**在进行推理设备选择的时候，我们应该综合考虑速度和价格成本**：

* 不同特点/业务的模型在不同的设备上的性价比不一样。
* 推理实例的带宽越大，同样payload大小的请求和响应的收发耗时就越小（SageMaker Endpiont的cloudwatch指标overhead latency包括了这个耗时）。
* 训练时使用GPU，推理时完全可以使用CPU（只要纯推理延迟能满足要求）。
* 如果整个推理链路性能的瓶颈或者说latency的main part并不是纯推理时间，那么一般可以考虑用CPU做推理，这样性价比更高。
* 模型的推理本身是计算消耗型的，因此如果选择CPU实例的话，需要选择计算优化型的而不是内存优化型的（发现经常有客户选用M5/M4系列而不是C5系列，他们的推理速度差别挺大的）

1. **推理框架本身的优化**

推理框架本身能提供的一些优化也是非常重要的，这里的优化指的是框架本身的实现的高效性以及提供的一些可以配置的优化选项。当前业界比较成熟的推理框架是比如Triton，Tensorflow Serving（简称TFS）等。当使用GPU实例做推理时，可以优先尝试Triton on SageMaker；还可以尝试先用TensorRT对模型做编译，然后用Triton on SageMaker做serving。

举例：对于TFS，如果在CPU实例上进行推理，可以通过设置如下的参数来加速（这些相关的参数或者环境变量如何设置，之后会详细介绍）。

* TFS inter op和intra op的并行度设置
* MKL-DNN的相关设置比如线程池数量，以及线程与CPU的bind策略

总结

本篇文章从相对宏观的角度来介绍了在SageMaker中的机器学习推理，相信大家已经对SageMaker在推理场景提供的能力有了大致的了解，也对SageMaker上进行推理优化有了一定的的认识。接下来我会用另一个文章来介绍一个具体的场景，即SageMaker 模型部署服务中的内置TFServing模型推理优化。