计 算 机 研 究 与 发 展
Journal of Computer Research and Development

https://crad.ict.ac.cn
62(2): 503−516, 2025

基于概率分布的无服务器计算弹性伸缩算法

李 威[1] 李光辉[1] 赵庆林[2] 代成龙[1] 陈 思[1]

[1](江南大学人工智能与计算机学院 江苏无锡 214122)

[2](澳门科技大学计算机科学与工程学院 澳门 999078)

(6213113078@stu.jiangnan.edu.cn)

Probability Distribution Based Auto-Scaling Algorithm in Serverless Computing

Li Wei[1], Li Guanghui[1], Zhao Qinglin[2], Dai Chenglong[1], and Chen Si[1]

[1](*School of Artificial Intelligence and Computer Science*, *Jiangnan University*, *Wuxi*, *Jiangsu* 214122)

[2](*School of Computer Science and Engineering*, *Macau University of Science and Technology*, *Macau* 999078)

Abstract Serverless computing provides developers a cloud computing paradigm, which does not require that developers focus on the server operation and hardware resource management in the context of the popularity of container technology and micro-service framework. At the same time, serverless computing can adapt to dynamic load changes in real time through elastic expansion and contraction, which can effectively reduce the request response delay and the service cost, and meet the customer's demand for pay-as-you-go cloud service expense. However, serverless computing faces the issue of cold start delay caused by the demand for elastic expansion and contraction. Creating the instances of warm-up function in advance can reduce the frequency and delay of cold start effectively. Nevertheless, the traffic burst problem in the cloud environment greatly increases the difficulty of predicting the number of warm-up function instances. To solve the above-mentioned challenges, a probability distribution based auto-scaling algorithm (PDBAA) is proposed. By using the historical data of monitoring indicators to predict the probability distribution of future requests, the optimal number of warm-up function instances is calculated for minimizing the request response delay. PDBAA can effectively combine the powerful prediction capability of deep learning technology to further improve performance. Under the Knative framework, the performance of PDBAA is verified by NASA and WSAL datasets. The simulation results show that, compared with the Knative auto-scaling algorithm and other prediction algorithms, PDBAA improves the elastic performance by over 31%, and reduces the average response time by over 16%, which can better solve the traffic burst problem, and effectively reduce the response delay of serverless computing requests.

Key words serverless computing; function-as-a-service; cold start delay; auto-scaling; deep learning

摘 要 在容器技术和微服务框架的普及背景下,无服务器计算为开发者提供了一种无需关注服务器操作以及硬件资源管理的云计算范式.与此同时,无服务器计算通过弹性扩缩容实时地适应动态负载变化,能够有效降低请求响应延时并且减少服务成本,满足了客户对于云服务成本按需付费的需求.然而,无服务器计算中面临着弹性扩缩容需求导致的冷启动延迟问题.提前预热函数实例能够有效地降低冷启动发生频率和延时.然而,在云环境中流量突发问题极大地增加了预测预热函数实例数的难度.针对上述挑战,提出了一种基于概率分布的弹性伸缩算法(probability distribution based auto-scaling algorithm, PDBAA),利

收稿日期: 2023-03-27; **修回日期**: 2024-05-20

基金项目: 国家自然科学基金项目(62072216)

This work was supported by the National Natural Science Foundation of China (62072216).

通信作者: 李光辉 (ghli@jiangnan.edu.cn)

用监控指标历史数据预测未来请求的概率分布,以最小化请求响应延时为目的计算预热函数实例的最佳数量,并且PDBAA能够有效地结合深度学习技术的强大预测功能进一步提升性能.在Knative框架中,通过NASA和WSAL数据集对算法进行了验证,仿真实验表明,相比于Knative弹性伸缩算法以及其他预测算法,所提出的算法弹性性能提升了31%以上,平均响应时间降低了16%以上,能够更好地解决流量突发问题,有效地降低了无服务器计算请求的响应延时.

关键词 无服务器计算;函数即服务;冷启动延时;弹性伸缩;深度学习

中图法分类号 TP311

DOI: 10.7544/issn1000-1239.202330191 **CSTR:** 32373.14.issn1000-1239.202330191

无服务器计算作为云计算模型的新范式,由于其管理简单和轻量级的特性受到了极大的欢迎.在无服务器计算中,开发人员无需管理服务器的基础设施资源,只需要编写云函数以及设置几个简单的参数,上传到无服务器平台,便可以使用相应的接口或者HTTP请求调用部署的函数.与传统服务器计算模型不同的是,无服务器计算旨在简化开发人员对服务器的管理操作,让用户更容易使用云服务,同时提供现收现付成本模型节省成本,而不是基于预订的模型[1].据云原生计算基金会(cloud native computing foundation, CNCF)定义,无服务器计算是指在构造和运行应用程序时无需管理服务器的一种计算范式.它描述了细粒度部署模型,由1个或多个函数组成的应用可上传到平台,并执行、扩缩容和对基于实际运行时的资源消耗进行计费[2].

根据云原生计算基金会对无服务器计算的定义,无服务器计算带来的主要优势为:1)可伸缩性,无服务器计算增加和释放计算资源不依赖运维团队,部署的服务能够在流量峰值到来时快速、自动地扩展,反之,能在流量减少时自动释放资源降低成本;2)现收现付计费模式,在无服务器计算中,用户只需为使用的计算资源付费,即只在执行云函数时才付费.

虽然无服务器计算有许多特点和优势,但与传统服务器计算之间的差异将带来许多挑战.这些挑战会对系统产生性能和安全等方面的巨大影响.在无服务器计算中,函数在无服务器平台上必须完成初始化工作才能执行.一旦函数被调用,第1步工作就是准备一个函数实例,是否创建新函数实例响应函数请求是冷启动和热启动的主要区别.通常来说,冷启动的延时要远远高于热启动[3].无服务器计算希望给用户提供一种高质量且低成本的服务,因此,如何以尽可能低的成本降低冷启动发生率是无服务器计算主要的性能挑战.

尽管现有的冷启动优化技术(如缓存优化、实例复用、实例预热等)在多数情况下都是有效的,但在处理冷启动问题时,它仍会遇到一些性能问题.一方面,这些冷启动优化技术会带来额外的资源消耗或者计算时延.另一方面,一些延迟远远高于并发应用程序中的平均值,称为尾部延迟,由函数初始化、存储访问和突发函数调用流量造成.在实例预热技术中,通常采用时间序列预测的方法,例如移动平均算法、指数平滑法、ARIMA等.此外,由于近年来深度学习和强化学习的快速发展,许多研究人员将深度学习算法和强化学习算法应用到实例预热技术中.然而,在云环境中存在的流量突发问题极大地增加了预测预热函数实例数量的难度,导致当出现流量突发情况时,集群中的函数实例难以及时响应请求流量而造成请求堵塞,这对低延时要求的服务是致命的.

为了解决以上问题,本文研究了流量突发情况下的数据特性,使用提前预热函数实例的方法解决冷启动问题.本文的主要贡献有3点:

1)提出了一种基于概率分布的弹性伸缩算法(probability distribution based auto-scaling algorithm, PDBAA),以最小化请求响应时间为目标,有效降低了冷启动问题对无服务器计算的影响.

2)将预热函数实例数量预测问题分解为2个子问题:概率分布预测和预热函数实例数量计算.针对概率分布预测子问题,利用历史数据构造未来请求的概率分布.此外,还能有效地结合深度学习模型预测未来请求的概率分布;针对预热函数实例数量计算子问题,利用数据的离散程度计算最佳预热函数实例数量,使得函数实例数能够快速响应请求,避免函数冷启动发生,有效地降低了请求响应时间.

3)在Knative框架中,通过仿真实验验证PDBAA算法的性能.将PDBAA算法与其他基准算法进行对比,实验结果表明PDBAA算法可以有效地降低请求延时以及SLA违约率.在2个真实数据集上,PDBAA

算法与 KPA 算法相比，弹性性能提升了 31% 以上，平均响应时间降低了 16% 以上. 此外，PDBAA 算法的预测时间开销很小，相比于 2 s 的弹性伸缩周期可忽略不计，可实现实时弹性伸缩.

1 相关工作

现有的冷启动解决方案大致分为 2 种思路：减少冷启动延迟时间（如缓存优化，又称温启动）和降低冷启动发生频率（例如实例复用、实例预热）.

1）减少冷启动延迟时间. 该方面的研究主要包括减少函数实例准备延迟以及将函数代码和其他依赖项预先加载到内存. 在减少容器准备延迟研究中，OpenWhisk 平台使用预热容器，并根据内存和编程语言对容器进行分类. 通过这种方法减少容器的准备时间，从而缓解冷启动延迟[4]. 文献 [5] 中提供一个名为 SAND 的平台，引入了一个应用程序级沙盒将应用程序的所有函数放置在同一容器上，通过允许同一沙盒中的实例共享代码库以降低容器准备延迟. 文献 [6] 中提供了一个名为 SOCK 的平台解决了容器中使用的 Linux 内核性能瓶颈并使用 Zygote 技术预导入函数所需的库. 文献 [7] 中介绍了一种称为预烘焙的方法，该方法可以从以前执行的函数中恢复快照. 文献 [8] 中提出自适应容器池伸缩策略（adaptive container pool scaling strategy, ACPS）. ACPS 有一个预先启动的容器池，根据不同的编程语言进行分类，且预测每个类别的冷启动次数，以自适应地确定容器实例数；在将函数代码和其他依赖项预先加载到内存的研究中，文献 [9] 利用包感知调度减少冷启动延迟，包感知调度是函数调度到已经缓存函数需要加载依赖包的工作节点上，并利用一致性哈希算法保证集群的负载均衡.

2）降低冷启动发生频率. 该思路的相关研究中主要有 2 种解决方案：提前预热函数实例和实例重用. 在提前预热函数实例研究中，CloudWatch[10]，Thundra. io[11]，Dashbird. io[12]，Lambda Warmer[13] 等工具定期调用函数并预热它们（如每 5 分钟 1 次）. 文献 [14] 中使用 Q-learning 算法动态确定函数实例的数量. 可用函数实例的数量和每个函数的离散化 CPU 利用率被视为环境状态（state），动作（action）设定为放大或缩小函数实例. 文献 [15] 中使用深度学习 LSTM 模型预测工作负载，并提出逐步减少策略避免当伸缩操作过于频繁时发生振荡；在实例重用相关研究中，在一些无服务器计算平台中，如 Google Cloud Functions, AWS Lambda, Microsoft Azure, Open Lambda, OpenWhisk，在函数执行完成后，函数实例及其资源会保留一段时间，因此在该时间段收到的请求，将被函数实例以热启动方式响应. 具体地说，在 OpenWhisk 平台中，函数实例在执行后 50 ms 内保持运行或热状态. 在此之后，函数实例及其资源将被保留，10 min 后才完全释放，这一时间被称为空闲窗口. 这种方法能有效地减少延迟，因为重新启动保留资源的容器延迟比分配新容器延迟更少. 此外，AWS Lambda 平台提供了 Provisioned Concurrency 机制，负责支持要求延迟更低且更具可预测性的工作负载，开发人员可以配置 Provisioned Concurrency 为服务提供良好的冷启动回避效果[16]. 文献 [17] 中提出了预热与重用相结合的策略，该策略考虑到函数在调用模式上的异构性，采用直方图对函数的调用方式进行分析. 该策略使用一个依赖于具体函数预热窗口（指上一次函数执行完成后，间隔多长时间再次进行预热）参数和保活窗口（指容器空闲后维持多长时间不释放）参数分别对预热和重用过程进行控制. 针对 3 种不同的调用模式分别采用不同的策略. 对于调用模式非常规律的情况，采用一个较长的预热窗口和一个较短的保活窗口；对于调用模式不规律的情况，则采用固定长度的保活窗口；对于调用频率较低的应用，其到达时间间隔会超出直方图的表示范围，因此采用 ARIMA 模型对达到时间进行预测. 文献 [18] 中提供了一个名为 ETAS 的函数调度器，ETAS 根据函数的执行时间、到达时间和容器状态预测函数调用的时间，并根据函数实例的预测信息为其分配最佳工作节点. 文献 [19] 中提出了一种容器生命周期感知调度，称为 CAS. 关键思想是根据请求的分布和容器的不同生命周期阶段创建或驱逐容器. 文献 [20] 中提出利用强化学习算法计算预热实例的最佳时间以及基于 LSTM 预测未来的函数调用时间以确定所需的预热实例数.

在这些方法中，预热实例、暂停和重用实例、创建热实例池以及调用函数等过程会造成资源浪费，其原因是存在当实例保持预热状态时没有请求访问的情况. 另一方面，考虑到云环境的动态性，函数调用随时间的变化是不能忽略的，故提供固定机制以减少冷启动延迟的方法不能适应云环境的动态性. 由于云环境的动态性，无服务器计算的弹性伸缩机制需要一种自适应策略，通过监控云环境提供自动化决策. 因此，基于强化学习和深度学习的算法很适用于这一领域. 经典的强化学习算法需要环境和动作空间具有离散性，但在无服务器计算弹性伸缩中，

环境状态如内存、CPU 利用率以及调用时间等都具有连续性. 如果使用深度学习算法预测预热实例数, 由于在云环境的突发性, 很难在简单的模型下具有高准确性, 但使用复杂的深度学习模型预测则会导致高昂的计算成本以及时间开销. 因此, 本文面向无服务器计算的弹性伸缩系统, 利用历史数据的离散程度, 就如何解决流量突发问题和实现低时间复杂度的弹性伸缩算法展开研究.

2 模型设计

2.1 应用场景分析

如图 1 所示, 无服务器计算集群包含了 1 个主节点和若干个工作节点, 为若干个函数服务提供硬件资源, 将应用逻辑划分为独立的函数. 开发人员编写函数的代码, 其中每个函数负责处理一个特定的任务, 并使用无服务器平台提供的工具进行部署. 无服务器计算平台会提供相应的函数调用接口, 用户通过平台提供的相应接口调用函数服务. 弹性伸缩器将监控用户请求的流量, 通过相应的弹性伸缩算法计算函数实例数, 扩容的函数实例通过调度器根据相应的调度算法调度到相应的节点上进行创建, 以及时处理各种波峰波谷及平稳负载.

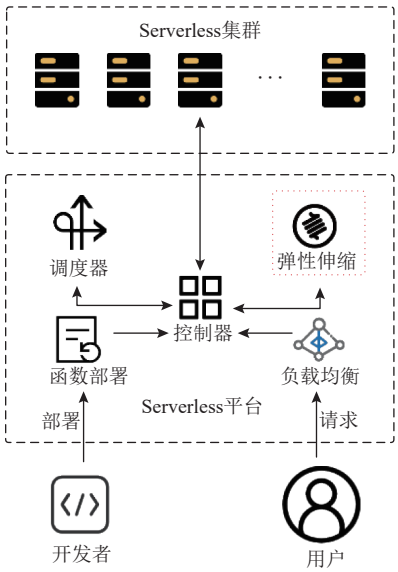


Fig. 1 Application scene for auto-scaling in serverless computing

图 1 无服务器计算弹性伸缩应用场景

2.2 Knative Pod Autoscaler

Knative 是一款由 Google 发布的基于 Kubernetes 的 Serverless 开源框架[21], Knative 由 Eventing 和 Serving 组成. Eventing 组件针对 Serverless 事件驱动模式做了一套完整的设计, 包括外部事件源的接入、事件注册和订阅, 以及对事件的过滤等功能. Serving 组件的职责是管理 Pod 和工作负载, Pod 是 Kubernetes 中创建和管理的最小单元, 由一组容器组成. Serving 组件的功能包括弹性伸缩、服务路由及流量控制等.

Knative Serving 为 Knative 提供了至关重要的弹性伸缩机制, 并且支持缩容到零. 在 Knative 中有 3 个关键组件支持 Knative 的弹性伸缩机制: 激活器(Activator)、弹性伸缩器(Autoscaler)和 Queue Proxy. Activator 组件负责为不活跃状态的 Revision 接收并缓存请求, 同时报告指标数据给弹性伸缩器, 在弹性伸缩器创建 Revision 之后, 它还负责将请求发送到 Revision. 弹性伸缩器组件接收请求指标数据并调整需要的 Pod 数量以处理流量负载. 在 Knative 中 Pod 通常由 2 个容器组成, 分别是 User Container 和 Queue Proxy Container. User Container 为业务容器, 包含应用程序及其依赖的运行环境. Knative Serving 为每个 Pod 注入 Queue Proxy Container, 该容器负责向弹性伸缩器报告 User Container 的并发指标. 弹性伸缩器接收到数据后根据并发请求数及相应的算法, 调整部署器(Deployment)的 Pod 数量, Knative 弹性伸缩具体过程如图 2 所示.

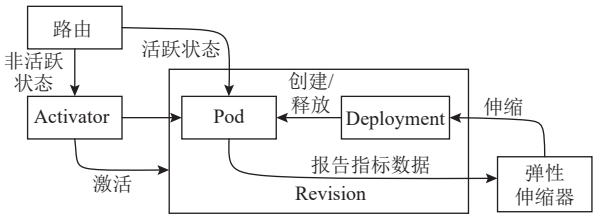


Fig. 2 Knative auto-scaling architecture

图 2 Knative 弹性伸缩架构

Autoscaler 基于每个 Pod 的平均请求数或并发数进行自动扩缩容, 按照式(1)计算 Pod 数量:

$$NewOrderReplica = \left\lceil \frac{OV}{TV} \right\rceil, \tag{1}$$

其中监测值(observed value, OV)和目标值(target value, TV)分别是监控指标值和用户设置的目标指标值. 默认情况下, Knative 弹性伸缩每 2 秒进行 1 次, 然后相应地在 Kubernetes 集群中创建或释放 Pod. 在 Knative 中, OV 的计算使用移动平均算法, 默认使用 60 s 的数据计算平均值, 以生成稳定的 Pod 数.

在 Knative 中使用的基于请求数的弹性伸缩方法中, 目前有 2 种广泛使用的指标: 并发值和每秒请求数. 这些指标中的任何一个都可以用作主要监控指标, 并将与 Pod 数计算的目标值比较进行伸缩. 这些指标将由 Knative 注入 Kubernetes 部署的 Queue Proxy

Container 监控并每秒收集 1 次.

除此之外, Knative 会考虑更多的场景, 例如流量突发场景. KPA 为此设计了 Stable 和 Panic 模式, 目的是避免流量突发时扩容不及时, 在 Knative 中 Stable 时间窗口默认值为 60 s, Panic 时间窗口默认值为 6 s. 除了 Autoscaler 会进入 Panic 模式更迅速地扩容外, Knative 中还有很多的配置用于处理突发流量的场景. 其中一个很重要的参数目标利用率(target utilization)为在计算 Pod 数时, *TV* 会乘以目标利用率, 在 Knative 中目标利用率默认值为 0.7.

在 Knative 中, 为了解决流量突发的场景, 设计了目标利用率并将其设置为 0.7, 该方法一定程度上能解决流量突发的场景. 但该参数值设置为一个固定值难以适应云环境的动态性, 如没有出现流量突发的场景会造成成本的增加. 此外, 流量突发的场景中, 固定的目标利用率会导致无法满足扩容需求.

2.3 PDBAA 算法

无服务器计算的弹性伸缩问题是一个经典的自动控制问题, 它要求弹性伸缩器实时地调整函数实例数量以响应动态负载变化, 其过程通常被抽象为监控、分析、计划和执行(MAPE)控制循环[22]. PDBAA 模型如图 3 所示, 当 Revision 中没有就绪 Pod 时(非活跃状态), 客户请求通过网关转发到 Activator, 然后由 Activator 接收并缓存请求并向弹性伸缩器的监控模块报告请求量. 反之, 当 Revision 处于活跃状态时, 客户请求直接转发到 Pod 的 User Container 中, 弹性伸缩器的监控模块会向 Queue Proxy Container 收集指标, 分析模块利用历史数据计算请求的概率分布, 再结合离散程度计算 *OV* 值, 计划模块计算利用 *OV* 值相应的预热 Pod 数, 执行模块根据预热 Pod 数进行相应的伸缩操作.

监控模块持续收集分析和计划模块所需的不同类型的数据, 以确定适当的伸缩操作. 在所呈现的架构中, 从 2 个不同的来源收集具有各种特征的数据: 1)请求数据, 如每秒请求数、并发量; 2)集群中所有 Pod 状态. 将收集的历史数据经过处理再整合相关的时间戳构建成时间序列数据, 该时间序列数据将用于计算预热 Pod 数.

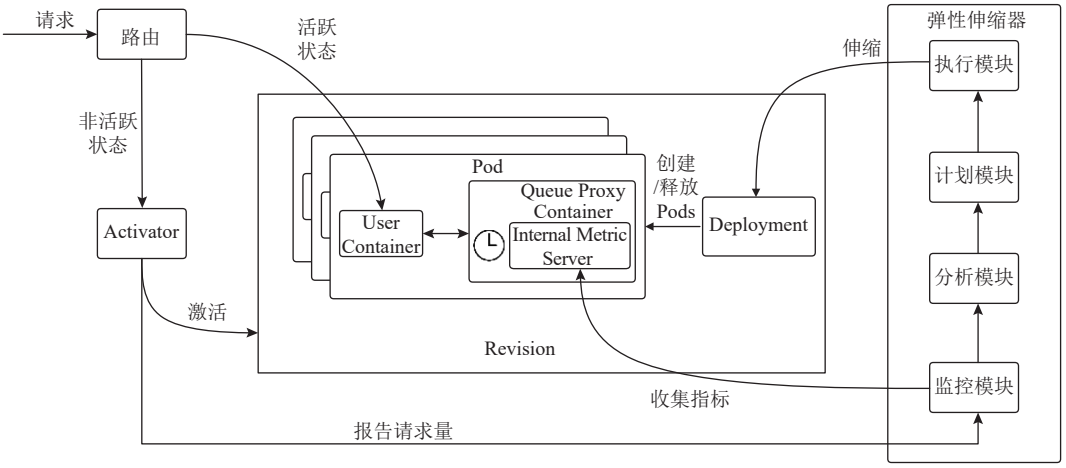


Fig. 3 PDBAA model frame

图 3 PDBAA 模型框架

在分析模块中, 利用上一阶段采集的历史数据计算 *OV* 值. 模型期望计算的 *OV* 值为下一个时间周期中的最大负载值, 目标是最小化请求的响应时间. 然而, 在流量突发情况下, 会收到不平稳的数据, 时间序列预测方法很难直接准确地预测预热 Pod 数量. 因此, 本文将预热 Pod 数量预测问题分解为 2 个子问题: 概率分布预测和预热 Pod 数量计算.

首先利用收集的请求数据计算历史数据的平均值 μ 和方差 σ, 利用式(2)计算泊松分布的概率密度函数, 泊松分布适合描述无服务器计算的请求数据分布, 在文献 [23–24] 中均采用了泊松分布, 利用历史数据计算出的概率分布很大程度上能代表下一个时间周期的数据概率分布:

$$f(x)=\frac{\mu^x}{x!}\exp(-\mu),\ \ x=0,1,\cdots. \tag{2}$$

为了简化泊松分布的概率计算, 计算概率时使用式(3):

$$f(x)=\begin{cases}\exp(-\mu), & x=0,\\ \dfrac{\mu}{x}f(x-1), & x>0.\end{cases} \tag{3}$$

由概率密度函数求得累积概率分布:

$$F(x)=P\{X\leqslant x\}=\int_{-\infty}^{x}f(t)\mathrm{d}t. \tag{4}$$

再利用累积概率分布求得 OV 值:

$$OV=F^{-1}(p), \tag{5}$$

其中 p 代表置信概率, 即未来数据值都包含在某一范围内的概率. 这表明计算得出的 Pod 数量很大概率能够处理未来的最大请求负载, 其计算公式为:

$$p=\alpha+(1-\alpha)\times\tanh(cv). \tag{6}$$

函数 tanh 图像如图 4 所示, 由于 cv 的取值范围为 $[0,+\infty)$, 利用函数 tanh 在定义域 $[0,+\infty)$ 区间上值域为 $[0,1)$, 单调递增的性质可以使得 p 和 cv 正相关, 且能保证 p 的取值范围为 $[0,1)$. α 表示当 $\sigma=0$ 时的置信概率, 当历史数据都为定值时, 得到的 OV 值会等于该值, α 的计算为

$$\alpha=F(\mu). \tag{7}$$

cv 为代表离散程度的系数, 即方差和平均值之比:

$$cv=\frac{\sigma}{\mu}. \tag{8}$$

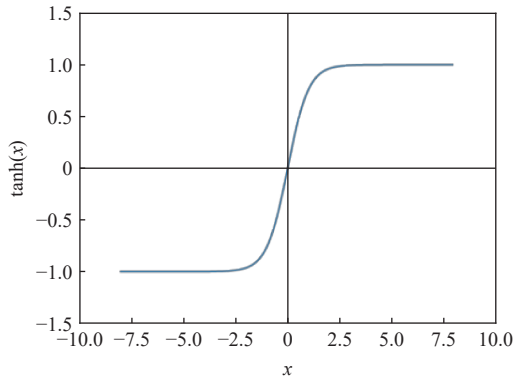


Fig. 4 Function tanh

图 4 函数 tanh

具体算法伪代码如算法 1 所示:

算法 1. PDBAA.

输入: 历史时间序列数据 $\boldsymbol{x}_t$;

输出: OV 值.

① $\mu=mean(\boldsymbol{x}_t)$, $\sigma=variance(\boldsymbol{x}_t)$; /*计算均值和方差*/

② if $\mu==0$ then/*均值为 0 时, 返回 OV 值为 0*/

③ return 0;

④ end if

⑤ $cv=\sigma/\mu$;

⑥ $p_\alpha=\exp(-\mu),\alpha=p_\alpha,x=0$;

⑦ for $x=1,2,\cdots,\lceil\mu\rceil$ do/*利用式(6)和式(3)计算 α*/

⑧ $p_\alpha=p_\alpha\times(\mu/x),\alpha=\alpha+p_\alpha$;

⑨ end for

⑩ $p=\alpha+(1-\alpha)\times\tanh(cv)$;

⑪ $p_{OV}=\exp(-\mu),F=p_{OV}$;

⑫ for $x=1,2,\cdots$ do/*利用式(5)和式(3)计算 OV*/

⑬ $p_{OV}=p_{OV}\times(\mu/x),F=F+p_{OV}$;

⑭ if $F\geqslant p$ then

⑮ return x;

⑯ end if

⑰ end for

在计算泊松分布的参数 μ 时, 不仅可以使用历史数据构造, 还可以使用深度学习模型预测. 本文设计的 PDBAA†算法结合基于长短期记忆(long short-term memory, LSTM)网络模型[25]以及 PDBAA 算法实现预热 Pod 数量的预测.

LSTM 是为了解决循环神经网络(recurrent neural network, RNN)模型梯度消失问题而改进的网络[26]. 如图 5 所示, LSTM 在 RNN 的基础上增加了 3 个门结构, 分别是输入门(input gate)、遗忘门(forget gate)和输出门(output gate).

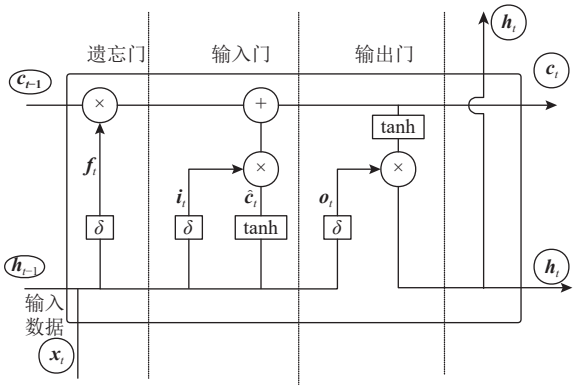


Fig. 5 LSTM structure

图 5 LSTM 结构

对于输入数据 $\boldsymbol{x}_t$, LSTM 每个时间周期都会更新记忆单元 $\boldsymbol{c}_t$ 和隐藏状态 $\boldsymbol{h}_t$. LSTM 单元的状态受输入 $\boldsymbol{x}_t$ 以及上一时刻的记忆单元 $\boldsymbol{c}_{t-1}$ 和隐藏状态 $\boldsymbol{h}_{t-1}$ 所影响. LSTM 单元更新机制如式(9)~(14)所示:

$$\boldsymbol{i}_t=\delta(\boldsymbol{W}_i[\boldsymbol{h}_{t-1},\boldsymbol{x}_t]+\boldsymbol{b}_i), \tag{9}$$

$$\boldsymbol{f}_t=\delta(\boldsymbol{W}_f[\boldsymbol{h}_{t-1},\boldsymbol{x}_t]+\boldsymbol{b}_f), \tag{10}$$

$$\boldsymbol{o}_t=\delta(\boldsymbol{W}_o[\boldsymbol{h}_{t-1},\boldsymbol{x}_t]+\boldsymbol{b}_o), \tag{11}$$

$$\hat{\boldsymbol{c}}_t=\tanh(\boldsymbol{W}_c[\boldsymbol{h}_{t-1},\boldsymbol{x}_t]+\boldsymbol{b}_c), \tag{12}$$

$$\boldsymbol{c}_t=\boldsymbol{f}_t\otimes\boldsymbol{c}_{t-1}+\boldsymbol{i}_t\otimes\hat{\boldsymbol{c}}_t, \tag{13}$$

$$\boldsymbol{h}_t=\boldsymbol{o}_t\otimes\tanh(\boldsymbol{c}_t), \tag{14}$$

其中: $\boldsymbol{W}_i$, $\boldsymbol{W}_f$, $\boldsymbol{W}_o$, $\boldsymbol{W}_c$ 是门结构的权重矩阵; $\boldsymbol{b}_i$, $\boldsymbol{b}_f$, $\boldsymbol{b}_o$,

$\boldsymbol{b}_c$ 是门结构的偏置；⊗符号代表哈达玛积(Hadamard product)，即矩阵上对应位置的元素相乘；δ 符号为 sigmoid 激活函数.

最大化对数似然训练模型：

$$Loss = \ln p(\boldsymbol{x}_t|\theta(\boldsymbol{h}_t)). \tag{15}$$

使用平均值 μ 参数化泊松分布，使用 softplus 激活函数计算平均值，可以确保 $\mu \geqslant 0$，如式(16)所示：

$$\mu(\boldsymbol{h}_t) = \ln(1+\exp(\boldsymbol{W}_\mu \boldsymbol{h}_t + b_\mu)), \tag{16}$$

其中 $\boldsymbol{W}_\mu$, b_μ 分别是全连接层的权重和偏置.

网络模型结构如图 6 所示，将历史时间序列 $\boldsymbol{x}_t$ 作为网络模型的输入，通过 LSTM 层和线性层，再经过 softplus 激活函数得到模型的输出平均值 μ. 利用平均值 μ 构造出泊松分布，通过式(2)~(4)计算出累计概率分布，最后利用式(5)~(8)计算出 *OV* 值.

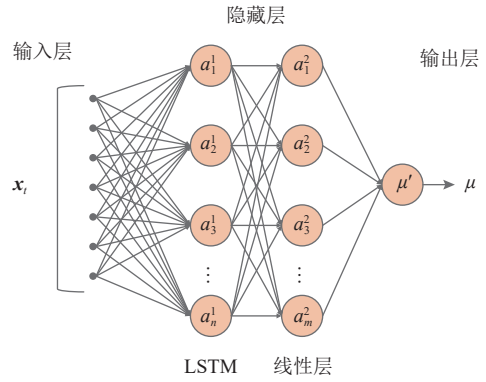


Fig. 6 Network model structure

图 6 网络模型结构

计划模块使用分析模块输出的 *OV* 值和 *TV* 值计算 *NewOrderReplica* 值，并将其与就绪 Pod 数量进行比较，以确定需要创建或释放的 Pod 数量. 具体来说，当 *NewOrderReplica* 值大于就绪 Pod 数量时，向执行模块发送扩容命令，反之亦然.

执行模块负责从计划模块接收命令并与 Kubernetes 通信，根据相应的伸缩命令伸缩 Pod 数量.

2.4 算法复杂度分析

在应对流量突发问题的解决办法中，KPA 算法设计了目标利用率参数并将其设置为 0.7，即计算的预热函数实例数除以 0.7. 该设计虽然一定程度上能缓解流量突发预测预热函数实例数的影响，但是会导致在流量平稳场景下存在冗余的函数实例. 此外，存在目标利用率参数设置为 0.7 时无法满足突发流量的情况，例如流量突发到平稳时的 2 倍，而 KPA 算法只能满足 1.43 倍的流量突发. 对于时间序列预测算法，在流量突发状况下，简单的时间序列预测算法如指数平滑算法、简单神经网络等准确性较低. 而使用复杂算法来预测突发流量虽然在准确性上有所提升，但其算法本身带来的时间成本会造成额外的延时，在无服务器计算场景下并不适用. 本文所提的 PDBAA 算法采用概率分布预测，并通过置信概率对突发流量的动态适用性来应对流量突发问题对预测预热函数实例数的影响. 具体而言，当出现流量突发趋势时，请求数据的平均值、方差以及离散系数均会上升，由式(4)~(6)可知，预热函数实例预测值与置信概率呈正相关，且置信概率与离散系数同样呈正相关，即预热函数实例预测值在流量突发时激增. 特别地，离散系数和流量突发的激烈程度呈正相关，即流量突发的趋势越大离散系数也越大，从而预测的预热函数实例数也越大，进而缓解流量突发所带来的请求堆积造成的严重冷启动延时问题. 此外，在流量平稳情况下，离散系数趋近于 0，PDBAA 算法计算的 *OV* 值等于请求数据的平均值，即预热函数实例数正好满足需求.

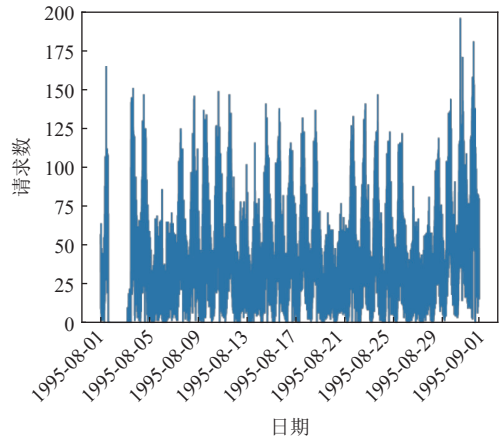
弹性伸缩算法需要实时满足动态负载，因此算法要具备时间复杂度低的特点，PDBAA 算法计算平均值 μ 和方差 σ 需要 $O(N)$ 时间复杂度，N 为历史数据的长度. 令 M 表示计算得到的 *OV* 值，且 M=*OV*，计算 *OV* 值的时间复杂度为 $O(M)$. 算法的整体时间复杂度为 $O(N+M)$.

3 实 验

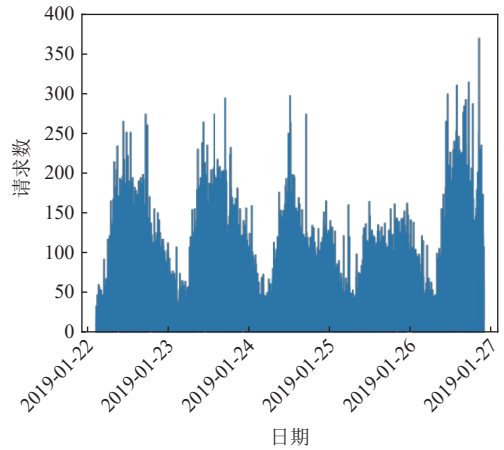
3.1 数据集与对比算法

为了评估所提出算法的性能，对不同的数据集和其他算法进行多方面的对比，评估了所提算法的弹性性能. 具体来说，本文使用 2 个真实的 Web 服务器日志评估所提出的算法：NASA Web 服务器日志[27]和伊朗电子商务 Web 服务器日志[28].

NASA Web 服务器日志由佛罗里达州的 NASA 肯尼迪航天中心服务器收集了 2 个月而得，分为 2 个子日志. 第 1 个子日志包含从 1995-07-01T00:00:00—1995-07-31T23:59:59(共 31 天)发出的所有 HTTP 请求. 第 2 个子日志包含从 1995-08-01T00:00:00—1995-08-031T23:59:59 发出的请求. 这 2 个子日志总共包含 3 461 612 条请求. 由于飓风艾琳，导致服务器关闭，因此在 1995-08-01T14:52:01—1995-08-03T04:36:13 之间没有日志. 通过将同一分钟内发生的所有日志累计到 1 个记录中预处理原始数据集，预处理后的数据集如图 7(a)所示. 因此，该数据集表示每分钟的请



(a) NASA数据集



(b) WSAL数据集

Fig. 7 Request load of different datasets

图 7 不同数据集的请求负载

求数. 将 1995-08-01T00:00:00—1995-08-24T00:00:00 的数据划分为训练集, 将 1995-08-24T00:00:00—1995-08-31T23:59:59 的数据划分为测试集.

WSAL(Web server access logs)是来自伊朗电子商务网站的日志, 包含 2019-01-22T03:56:14—2019-01-26T20:29:13 发出的所有 HTTP 请求. 这个日志总共包含 10 365 098 条请求. 通过将同一秒内发生的所有日志累积到 1 个记录中预处理原始数据集. 因此, 该数据集表示每秒请求数. 预处理后的数据集如图 7(b)所示. 将 2019-01-22T03:56:14—2019-01-25T00:00:00 的数据划分为训练集, 将 2019-01-25T00:00:00—2019-01-26T20:29:13 的数据划分为测试集.

对比算法如下:

1)KPA(Knative Pod AutoSacler). Knative 默认的弹性伸缩算法.

2)SES. 简单指数平滑(exponential smoothing)算法, 预测公式为

$$S_{t+1}^{(1)}=\alpha\times y_t+(1-\alpha)\times S_t^{(1)}, \tag{17}$$

其中$S_t^{(1)}$为时刻 t 的 1 次平滑值, α为加权系数(本实验设置为 0.5), y_t为时刻 t 的真实值.

3)DES. 2 次指数平滑(double exponential smoothing)算法, 预测公式为

$$S_{t+1}^{(2)}=\alpha\times S_t^{(1)}+(1-\alpha)\times S_t^{(2)}, \tag{18}$$

其中$S_t^{(2)}$为时刻 t 的 2 次平滑值.

4)TES. 3 次指数平滑(triple exponential smoothing)算法, 预测公式为

$$S_{t+1}^{(3)}=\alpha\times S_t^{(2)}+(1-\alpha)\times S_t^{(3)}, \tag{19}$$

其中$S_t^{(3)}$为时刻 t 的 3 次平滑值.

5)ARIMA. 文献 [29] 中提出的差分整合移动平均自回归(autoregressive integrated moving average)模型.

6)GDS. 文献 [15] 提出的逐步减少策略(gradually decreasing strategy), 使用 LSTM 模型预测工作负载以及提出逐步减少函数实例数策略以避免当伸缩操作过于频繁时发生的振荡.

7)BiLSTM. 文献 [30] 提出的应用预测模型, 用来计算预热函数实例数量, 使 Knative 更适应工作负载的变化, 以减少请求延迟时间.

3.2 实验描述

本文设计了 2 个实验评估所提出的弹性伸缩算法. 第 1 个实验的目的是评估模型的弹性伸缩性能. 第 2 个实验将在模拟环境中对系统架构和弹性伸缩器进行服务质量评估实验, 以研究平均响应时间和 SLA 违约率等性能指标. 其中 PDBAA†算法和 GDS 算法使用了 LSTM 模型, 其模型配置如表 1 所示.

Table 1 LSTM Model Parameters

表 1 LSTM 模型参数

参数	取值
输入大小	60
输出大小	1
LSTM 层数	3
LSTM 单元数	30
损失函数	MSE
优化器	Adam
批尺寸	32
训练轮次	20

在服务质量评估实验中, 利用 Python 的 SimPy 库实现了仿真环境, 以模拟 Knative 系统架构和弹性伸缩器 MAPE 循环. 如图 8 所示, 在 Serverless 中, Pod 实例是实际的资源消耗者亦是实际业务逻辑的执行者, 因此在概念上将 PodInstance 设计为一个 SimPy

中的进程. 此外, 触发器、弹性伸缩器、监控模块和创建函数实例函数同样被设计为 SimPy 进程. 模拟实验开始时, Trigger 进程不断地按照请求的发送时间将请求配置列表所描述的请求发送至集群, 直至所有的请求发送完毕, 触发器停止发送并销毁. 弹性伸缩器按照弹性伸缩时间周期不断地进行弹性伸缩, 直到仿真结束(当触发器被销毁即不会有新的请求到达且所有已发送的请求都执行完毕时). 监控模块按照监测时间周期不断地进行模拟状态的监测和记录. 仿真实验的主要参数如表 2 所示.

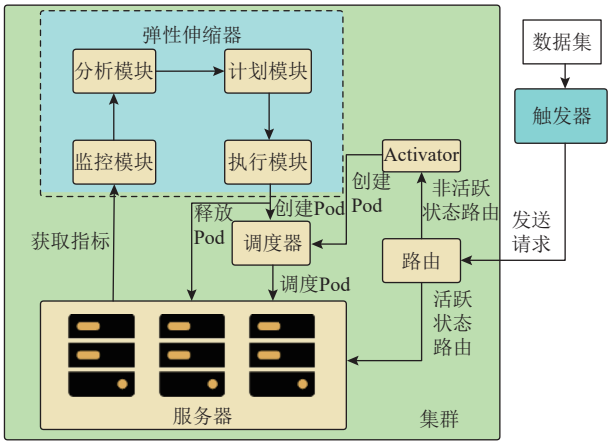


Fig. 8 Frame of simulation experiment

图 8 仿真实验框架

Table 2 Main Parameters Setting

表 2 主要参数设置

参数	数据集	
	NASA	WSAL
函数执行时间/s	0.2	0.2
Pod 创建时间/s	3	3
最大实例数	30	50
最小实例数	0	0
TV	5	5

3.3 性能指标

为了评估弹性伸缩器的供应精度和弹性增益, 本文使用了 SPEC(Standard Performance Evaluation Corporation)研究小组认可的面向系统的弹性指标[31], 如式 (22)~(24) 所示. r_t和p_t分别表示在时间 t 所需的资源和所提供的资源. Δt是在最近和当前所需资源或提供的资源变化之间的时间间隔.

对于资源配置的准确性, 资源配置不足指标 Θ_U 定义为满足服务级别目标(service-level objective, SLOs)所需的缺失 Pod 数量的相对占比. 过度供应指标 Θ_O 表示弹性伸缩器提供的冗余 Pod 数量的相对占比. Θ_U 和 Θ_O 的范围为 0~∞, 其中 Θ_U=Θ_O=0, 0 是最佳值. T_U 和 T_O 分别对应于当系统供应不足和过度供应时的时间占比. T_U 和 T_O 的范围为 0~100, 当测量时间内没有出现供应不足或过量供应时, T_U=T_O=0. 最后, 弹性增益ε_n是与其他弹性伸缩情况相比的弹性伸缩增益, 其中式(24)中 a 表示使用当前弹性伸缩算法, n 表示使用对比算法. 如果 ε_n>1, 则所提出的弹性伸缩算法相比所对比的弹性伸缩算法具有更优的弹性伸缩性能, 反之亦然.

$$\Theta_U = \frac{100}{T}\sum_{t=1}^{T}\frac{\max(r_t - p_t, 0)}{r_t}\Delta t, \tag{20}$$

$$\Theta_O = \frac{100}{T}\sum_{t=1}^{T}\frac{\max(p_t - r_t, 0)}{r_t}\Delta t, \tag{21}$$

$$T_U = \frac{100}{T}\sum_{t=1}^{T}\max(\mathrm{sgn}(r_t - p_t), 0)\Delta t, \tag{22}$$

$$T_O = \frac{100}{T}\sum_{t=1}^{T}\max(\mathrm{sgn}(p_t - r_t), 0)\Delta t, \tag{23}$$

$$\varepsilon_n = \left(\frac{\Theta_{U,n}}{\Theta_{U,a}}, \frac{\Theta_{O,n}}{\Theta_{O,a}}, \frac{T_{U,n}}{T_{U,a}}, \frac{T_{O,n}}{T_{O,a}}\right)^{\frac{1}{4}}. \tag{24}$$

此外, 为了评估无服务器计算弹性伸缩器的服务质量, 本文选用平均响应时间(请求提交时间到完成时间)和服务水平协议(service level agreement, SLA)违约率(请求响应时间超过 SLA 规定的预期时间, 本文实验设置该时间为 1 s)指标.

3.4 实验结果

1)弹性性能. 所提出的算法与其他对比算法在资源供应和弹性伸缩性能上进行了对比. 从表 3 中可以看出, 在弹性增益方面, PDBAA 算法优于其他算法, 在 NASA 数据集和 WSAL 数据集上的弹性增益值分别是 1.367 和 1.310, PDBAA 算法通过预测数据概率分布和置信概率, 提高对流量突发趋势的适应性. 这有效地解决了在流量突发状况下函数实例扩容不及时的问题, 因此相比于其他对比算法更准确地预测预热函数实例数, 从而得到更优的弹性性能. SES, DES, TES 算法的弹性增益值在 NASA 数据集略微低于 1, 分别为 0.941, 0.980, 0.981, 结果表明它们的弹性伸缩性能比 KPA 算法略差, 而在 WSAL 数据集上的弹性增益值分别为 1.022, 1.019, 1.005, 这代表它们在 WSAL 数据集上对比 KPA 算法略优, 整体来说这 3 种算法和 KPA 算法表现出相近的弹性伸缩性能. ARIMA 和 GDS 算法在 NASA 数据集上的弹性伸缩性能比 KPA 略优, 其弹性增益值分别为 1.066 和

1.162, 而在 WSAL 数据集上的性能有略微下降, 特别是 GDS 算法的弹性增益值只有 0.946, 其原因是 GDS 算法采用的深度学习算法对数据的稳定性具有一定的依赖. BiLSTM 算法相较于 KPA 算法在 NASA 数据集上表现出较差的弹性增益值, 仅为 0.863, 而在 WSAL 数据集上则表现出良好的弹性增益值, 达到了 1.059, 这是由于深度学习算法对数据较为敏感. 另一方面, 在资源配置不足指标 Θ_U 和 T_U 上, KPA 分别为 3.771% 和 17.607%. 所提出的 PDBAA 算法分别达到了 1.291% 和 5.470%, 在 WSAL 数据集上表现出相似的性能, 其 Θ_U 和 T_U 值分别为 2.385% 和 9.058%, 这表明 PDBAA 算法在 2 个数据集上都能够提供充足的 Pod 以响应请求. 由于资源调配不足会降低服务质量, 客户不倾向使用资源配置不足的平台, 供应商面临的挑战是确保在任何时间点都能提供足够的资源[32].

Table 3 Elasticity Performance Experimental Result

表 3 弹性伸缩性能实验结果

数据集	方法	Θ_O/%	Θ_U/%	T_O/%	T_U/%	ε_n
NASA	KPA	39.520	3.771	71.522	17.607	1.000
	SES	**36.506**	4.887	**68.328**	19.674	0.941
	DES	36.768	4.333	70.099	18.198	0.980
	TES	37.258	4.238	70.481	18.163	0.981
	ARIMA	44.528	2.908	76.194	14.742	1.066
	GDS	77.237	1.651	85.814	9.394	1.162
	BiLSTM	50.791	4.222	67.911	23.216	0.863
	PDBAA（本文）	62.059	**1.291**	84.108	**7.796**	**1.367**
	PDBAA†（本文）	75.137	**0.894**	90.780	**5.470**	**1.540**
WSAL	KPA	60.505	7.338	64.072	24.171	1.000
	SES	**46.290**	8.804	**57.228**	27.040	1.022
	DES	48.618	8.462	58.499	26.456	1.019
	TES	50.872	8.491	59.221	26.369	1.005
	ARIMA	80.902	5.617	72.544	20.116	1.009
	GDS	101.827	5.763	74.142	19.695	0.946
	BiLSTM	105.240	4.080	78.557	16.225	1.059
	PDBAA（本文）	125.734	**2.385**	85.830	**9.058**	**1.310**
	PDBAA†（本文）	143.374	**1.645**	90.855	**6.957**	**1.464**

注：加粗数值表示最优值.

值得一提的是, 结合 LSTM 模型的 PDBAA†算法能够预测概率分布, 进一步快速响应突发流量, 在 2 个数据集上其弹性增益值分别达到 1.540 和 1.464. 实验结果验证了 PDBAA†算法在响应突发流量上表现出优异的性能.

2)服务质量. 下面通过实验对各算法的服务质量进行评估. NASA 数据集的实验结果如图 9 所示, 在平均响应时间和 SLA 违约率方面, PDBAA 算法考虑了流量突发问题对预测预热函数实例数的影响, 能够在流量突发状况下准确预测预热函数实例数, 及时处理激增的请求流量, 因此表现出优异的性能, 平均响应时间和 SLA 违约率分别为 0.473 s 和 3.982%. GDS 算法凭借 LSTM 模型优秀的预测能力有效地降低了平均响应时间, 并且其逐步降低策略能缓解流量突发所造成的请求堆积问题, 平均响应时间和 SLA 违约率分别为 0.533 s 和 8.748%. BiLSTM 算法利用深度学习算法预测未来请求负载, 但深度学习算法在流量突发状况下难以准确预测, 因此相较于 KPA 算法较优, 平均响应时间和 SLA 违约率分别为 0.531 s 和 6.856%. 基准算法 KPA 得益于目标利用率参数的设计, 能够在一定程度上应对流量突发状况, 其平均响应时间和 SLA 违约率实验结果分别为 0.568 s 和 9.368%. 其次为 ARIMA, 该算法要求时序数据是稳定的且本质上只能捕捉线性关系, 因此很难解决流量突发问题, 平均响应时间和 SLA 违约率实验结果分别为 0.664 s 和 10.289%. SES 算法在该方面的性能最差, 平均响应时间和 SLA 违约率分别为 0.721 s 和 14.179%, DES 和 TES 是 SES 算法的再平滑, 相比于 SES 的性能较优, 平均响应时间分别是 0.687 s 和 0.664 s, SLA 违约率分别为 12.140% 和 11.228%.

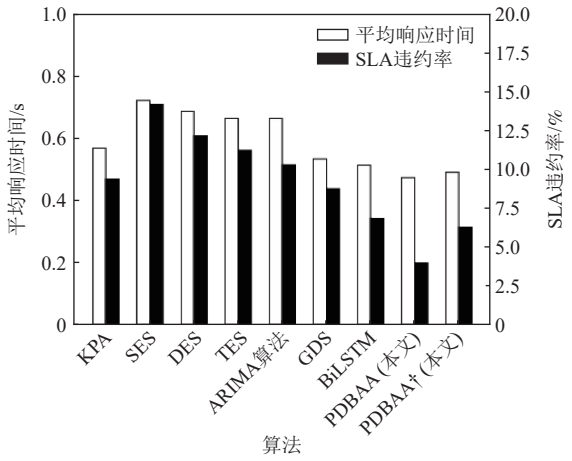


Fig. 9 Experimental results on NASA dataset

图 9 NASA 数据集上的实验结果

WSAL 数据集上的实验结果如图 10 所示, PDBAA 同样表现出最优的性能, 平均响应时间和 SLA 违约率分别为 0.586 s 和 9.091%. 算法的性能排名与 NASA 数据集几乎相同: PDBAA>GDS>BiLSTM>KPA>ARIMA>TES>DES>SES. 不同的是, 在 WSAL 数据集上 GDS 算法优于 BiLSTM 算法, 这是由于 WSAL 数据集的

数据特性导致 BiLSTM 的预测精度相较于 NASA 数据集降低，进而降低了服务质量. 此外，PDBAA†算法由于凭借神经网络模型的预测能力，在 NASA 数据集和 WSAL 数据集上表现出非常好的性能，在平均响应时间上分别为 0.491 s 和 0.584 s，SLA 违约率方面分别为 6.285% 和 10.850%. 与 KPA 算法相比，PDBAA 算法的平均响应时间在 NASA 数据集和 WSAL 数据集上分别降低了 16.7% 和 50.7%.

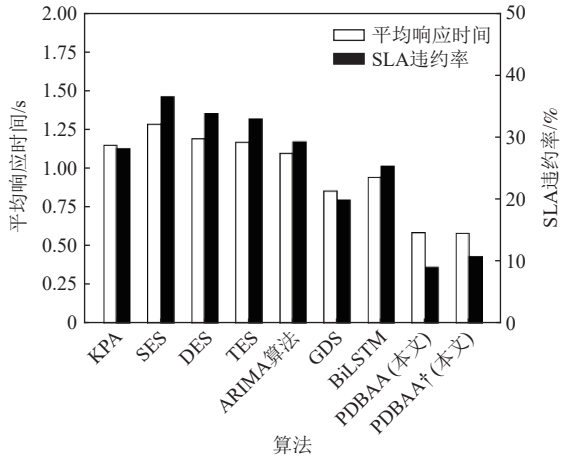


Fig. 10 Experimental results on WSAL dataset

图 10 WSAL 数据集上的实验结果

3)时间开销. 下面对各算法的计算速度进行了比较. 结果如表 4 所示. 可以看出 ARIMA 算法运行时间最长，达到了 656.3 ms，如果采用该算法进行弹性伸缩会造成巨大的时间开销，不利于实时满足动态负载. GDS, BiLSTM, PDBAA†算法都使用了深度学习算法，时间开销会相对较大，分别为 1.035 ms, 2.745 ms, 2.500 ms. 此外，KPA, SES, DES, TES, PDBAA 等算法的运行时间处于 0.040~0.070 ms 之间，这些算法都具有时间复杂度低的特点，非常适用于实时弹性伸缩.

Table 4 Running Time

表 4 运行时间 ms

方法	KPA	SES	DES	TES	ARIMA	GDS	BiLSTM	PDBAA（本文）	PDBAA†（本文）
运行时间	0.056	0.040	0.048	0.053	656.3	1.035	2.745	0.070	2.500

4)Pod 创建时间. 下面通过实验验证 Pod 创建时间对实验性能的影响. Pod 创建时间增加，即冷启动时延的增加，平均响应时间和 SLA 违约率都会随之增加. 在 NASA 数据集上 Pod 创建时间对平均响应时间的影响如图 11 所示，SES 算法的表现最差，随着 Pod 创建时间的增加，平均响应时间迅速增加. 对于 DES, TES, ARIMA 这 3 个算法，表现的性能略优于 SES，随着 Pod 创建时间的增加，平均响应时间增加不显著. 在 WSAL 数据集上 Pod 创建时间对平均响应时间的影响如图 12 所示，可以看出，PDBAA 和 PDBAA†算法随着 Pod 创建时间的增加，平均响应时间所受影响甚微. 而其他 5 种算法，随着 Pod 创建时间的增加，平均响应时间整体呈上升趋势. 其中，与 NASA 数据集相比，KPA 算法在 WSAL 数据集上表现出不同的性能，这是因为 WSAL 数据集的数据要比 NASA 数据集更不稳定，这证明了 KPA 算法在数据上呈现出一定程度的不稳定时性能会大幅度下降. 此外，GDS 算法在 Pod 创建时间大于 5.0 s 后平均响应时间剧增，Pod 创建时间为 8.0 s 时在 NASA 数据集和 WSAL 数据集上分别达到了 8.190 s 和 21.662 s. BiLSTM 算法和 KPA 算法表现出类似的性能，在 NASA 数据集上随着函数实例创建时间的增长，平均响应时间变化不大，而在 WSAL 数据集上性能则大幅度下降.

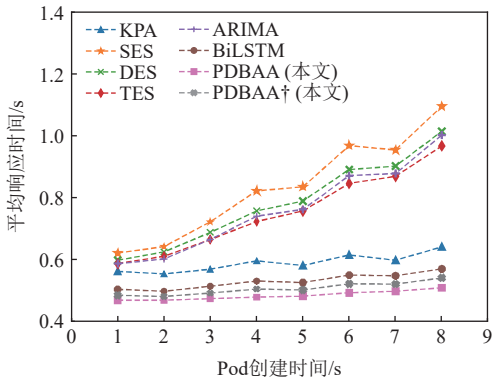


Fig. 11 Impact of Pod creation time on average response time on NASA dataset

图 11 Pod 创建时间在 NASA 数据集对平均响应时间的影响

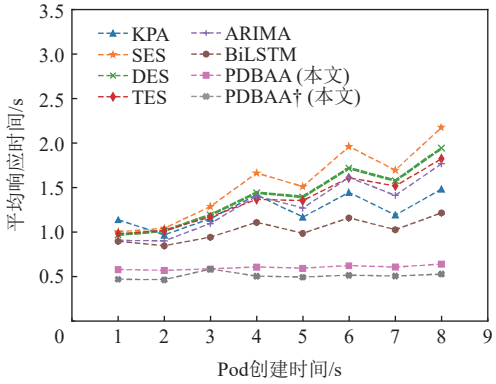


Fig. 12 Impact of Pod creation time on average response time on WSAL dataset

图 12 Pod 创建时间在 WSAL 数据集对平均响应时间的影响

在 NASA 数据集上 Pod 创建时间对 SLA 违约率的影响如图 13 所示, KPA 算法通过平均值计算预热函数实例数, 平均值对未来周期内的数据均具备指导意义, BiLSTM 在 NASA 数据集数据相对较稳定分布中预测能力较好, PDBAA 和 PDBAA†算法预测的概率分布对函数实例创建时间不敏感, 因此 KPA, BiLSTM, PDBAA, PDBAA†算法随着 Pod 创建时间的增加, SLA 违约率上升不明显. 对于其他算法, 当 Pod 创建时间增加时, 发生冷启动时堵塞的请求数量增加, 导致请求的 SLA 违约率呈现上升趋势. 在 WSAL 数据集上 Pod 创建时间对 SLA 违约率的影响如图 14 所示, 各算法受 Pod 创建时间的影响程度与 NASA 数据集的结果相似. 不同的是在 NASA 数据集上 PDBAA 算法要优于 PDBAA†算法, 而在 WSAL 数据集上则相反. 此外, 在 WSAL 数据集上当 Pod 创建时间为 3.0 s 时, PDBAA 算法要优于 PDBAA†算法.

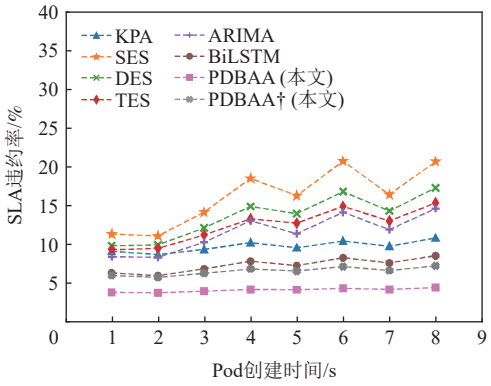


Fig. 13 Impact of Pod creation time on SLA default rate on NASA dataset

图 13 Pod 创建时间在 NASA 数据集对 SLA 违约率的影响

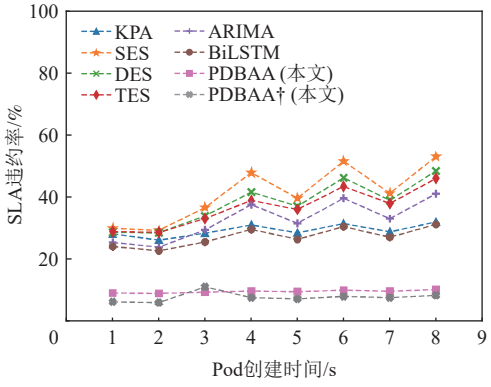


Fig. 14 Impact of Pod creation time on SLA default rate on WSAL dataset

图 14 Pod 创建时间在 WSAL 数据集对 SLA 违约率的影响

由于 PDBAA 算法和 PDBAA†算法通过预测数据的概率分布来预测预热函数实例数量, 数据的概率分布变化相较于单个数据值较为缓慢, 因此随着函数实例创建时间的增长, PDBAA 算法和 PDBAA†算法相较于 GDS 等算法在平均响应时间和 SLA 违约率性能上所受影响甚微. 在无服务器计算集群中会存在成千上万的服务, 而每个服务对应的 Pod 创建时间都不尽相同, 这表明无服务器计算的弹性伸缩算法需要对 Pod 创建时间不具备较强的相关性.

5)突发负载响应速度. 预测工作负载生成的 Pod 数量与实际工作负载如图 15 所示. 由图 15 可知: 当发生流量激增情况时, 由于 PDBAA 算法通过概率分布的预测以及置信概率对流量激增程度的适应性更精确地对预热函数实例数量进行预测, 因此 PDBAA 算法对于各种流量激增程度都能迅速地扩容相应的函数实例数来及时响应负载变化. 而 KPA 算法, 虽然通过 target utilization 参数的设计使其具备一定程度的突发负载响应能力, 但一旦出现超过其响应能力的突发负载, 将导致函数实例扩容不及时, 造成请求堆积以及请求响应延时过高的问题, 因此 PDBAA 算法相比于 KPA 算法具有更优的突发负载响应速度. 当请求负载下降时, PDBAA 算法亦能相应地缩放 Pod 数量降低资源消耗.

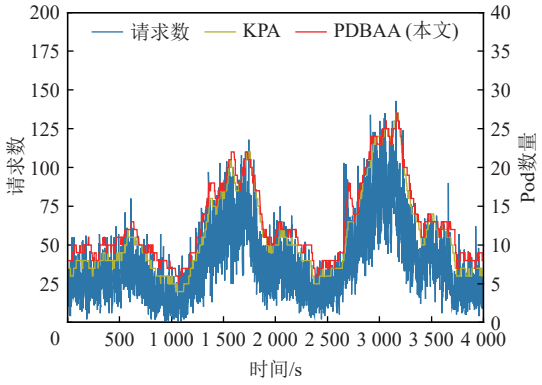


Fig. 15 Number of Pod in response to HTTP workload

图 15 响应 HTTP 工作负载的 Pod 数量

4 总 结

本文研究了无服务器计算中的冷启动延时问题, 为了最小化请求响应时间这一目标, 将预热函数实例数量预测问题分解为概率分布预测和预热函数实例数量计算 2 个子问题, 提出了一种基于概率分布的弹性伸缩算法, 并利用仿真实验评估所提出的算法性能. 实验结果表明, PDBAA 算法能最大限度降

低请求响应时间，优于 KPA 算法以及其他时间序列预测算法，能够有效降低 SLA 违约率，提升无服务器计算的服务质量. 未来工作中，将进一步研究在无服务器计算中保证服务质量的前提下以最小化服务成本为目标的弹性伸缩算法.

作者贡献声明：李威提出了算法思路和实验方案、完成实验并撰写论文；李光辉、赵庆林和代成龙提出了指导意见并修改论文；陈思协助完成实验.

参 考 文 献

[1] Schleier-Smith J, Sreekanti V, Khandelwal A, et al. What serverless computing is and should become: The next phase of cloud computing[J]. Communications of the ACM, 2021, 64(5): 76−84

[2] Castro P, Ishakian V, Muthusamy V, et al. The rise of serverless computing[J]. Communications of the ACM, 2019, 62(12): 44−54

[3] Shahrad M, Balkind J, Wentzlaff D. Architectural implications of function-as-a-service computing[C]//Proc of the 52nd Annual IEEE/ACM Int Symp on Microarchitecture. New York: ACM, 2019: 1063−1075

[4] Apache Software Foundation. Open source serverless cloud platform[EB/OL]. (2022−06−08)[2023−03−18]. https://openwhisk.apache.org/

[5] Akkus I E, Chen Ruichuan, Rimac I, et al. SAND: Towards high-performance serverless computing[C]//Proc of the 2018 USENIX Conf on USENIX Annual Technical Conf. Berkeley, CA: USENIX Association, 2018: 923−935

[6] Oakes E, Yang L, Zhou D, et al. SOCK: Rapid task provisioning with serverless-optimized containers[C]//Proc of the USENIX Annual Technical Conf. Berkeley, CA: USENIX Association, 2018: 57−70

[7] Silva P, Fireman D, Pereira T E. Prebaking functions to warm the serverless cold start[C]//Proc of the 21st Int Middleware Conf. New York: ACM, 2020: 1−13

[8] Xu Zhengjun, Zhang Haitao, Geng Xin, et al. Adaptive function launching acceleration in serverless computing platforms[C]//Proc of the 25th IEEE Int Conf on Parallel and Distributed Systems. Piscataway, NJ: IEEE, 2019: 9−16

[9] Aumala G, Boza E, Ortiz-Avilés L, et al. Beyond load balancing: Package-aware scheduling for serverless platforms[C]//Proc of the 19th IEEE/ACM Int Symp on Cluster, Cloud and Grid Computing. Piscataway, NJ: IEEE, 2019: 282−291

[10] Amazon Web Services, Inc. Application and infrastructure monitoring−Amazon CloudWatch−Amazon Web services[EB/OL]. (2022-09-16)[2023-03-18]. https://aws.amazon.com/cloudwatch/

[11] Thundra, Inc. Monitor, debug, test microservices on the cloud[EB/OL]. (2022-08-21)[2023-03-18]. https://www.thundra.io/

[12] Dashbird. Monitor serverless AWS applications at any scale-Dashbird[EB/OL]. (2022-12-16)[2023-03-18]. https://dashbird.io/

[13] Jeremy D. A module to optimize AWS Lambda function cold starts[EB/OL]. (2022-10-06)[2023-03-18]. https://github.com/jeremydaly/lambda-warmer

[14] Agarwal S, Rodriguez M A, Buyya R. A reinforcement learning approach to reduce serverless function cold start frequency[C]//Proc of the 21st IEEE/ACM Int Symp on Cluster, Cloud and Internet Computing. Piscataway, NJ: IEEE, 2021: 797−803

[15] Imdoukh M, Ahmad I, Alfailakawi M G. Machine learning-based auto-scaling for containerized applications[J]. Neural Computing and Applications, 2020, 32(13): 9745−9760

[16] Amazon Web Services, Inc. AWS Lambda announces provisioned concurrency[EB/OL]. (2022-07-12)[2023-03-18] https://aws.amazon.com/about-aws/whats-new/2019/12/aws-lambda-announces-provisioned-concurrency/

[17] Shahrad M, Fonseca R, Goiri Í, et al. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider[C]//Proc of the 2020 USENIX Annual Technical Conf. Berkeley, CA: USENIX Association, 2020: 205−218

[18] Banaei A, Sharifi M. ETAS: Predictive scheduling of functions on worker nodes of Apache OpenWhisk platform[J]. The Journal of Supercomputing, 2022, 78(4): 5358−5393

[19] Wu Song, Tao Zhiheng, Fan Hao, et al. Container lifecycle - aware scheduling for serverless computing[J]. Software: Practice and Experience, 2022, 52(2): 337−352

[20] Vahidinia P, Farahani B, Aliee F S. Mitigating cold start problem in serverless computing: A reinforcement learning approach[J]. IEEE Internet of Things Journal, 2022, 10(5): 3917−3927

[21] Google Inc. Knative is an open-source enterprise-level solution to build serverless and event driven applications[EB/OL]. (2023-03-14)[2023-03-18]. https://knative.dev/docs/

[22] Qu Chenhao, Calheiros R N, Buyya R. Auto-scaling Web applications in clouds: A taxonomy and survey[J]. ACM Computing Surveys, 2018, 51(4): 1−33

[23] Mahmoudi N, Khazaei H. Performance modeling of metric-based serverless computing platforms[J]. IEEE Transactions on Cloud Computing, 2022, 11(2): 1899−1910

[24] Hancock R, Udayashankar S, Mashtizadeh A J, et al. OrcBench: A representative serverless benchmark[C]//Proc of the 15th IEEE Int Conf on Cloud Computing. Piscataway, NJ: IEEE, 2022: 103−108

[25] Hochreiter S, Schmidhuber J. Long short-term memory[J]. Neural computation, 1997, 9(8): 1735−1780

[26] Pascanu R, Mikolov T, Bengio Y. On the difficulty of training recurrent neural networks[C]//Proc of the 30th Int Conf on Int Conf on Machine Learning. New York: ACM, 2013: 1310−1318

[27] National Aeronautics and Space Administration. Two months of HTTP logs from the KSC-NASA WWW server[EB/OL]. (2020-08-01)[2023-03-18]. https://ita.ee.lbl.gov/html/contrib/NASA-HTTP.html

[28] Kaggle Inc. Web server access logs[EB/OL]. (2022-04-13)[2023-03-18]. https://www.kaggle.com/datasets/eliasdabbas/web-server-access-logs

[29] Ariyo A A, Adewumi A O, Ayo C K. Stock price prediction using the ARIMA model[C]//Proc of the 16th UKSim-AMSS Int Conf on Computer Modelling and Simulation. Piscataway, NJ: IEEE, 2014: 106−112

[30] Phung H D, Kim Y. A Prediction based autoscaling in serverless computing[C]//Proc of the 13th Int Conf on Information and Communication Technology Convergence. Piscataway, NJ: IEEE,

2022: 763−766
[31] Bauer A, Grohmann J, Herbst N, et al. On the value of service demand estimation for auto-scaling[C]//Proc of the 19th Int Conf on Measurement, Modelling and Evaluation of Computing Systems. Berlin: Springer, 2018: 142−156
[32] Herbst N, Krebs R, Oikonomou G, et al. Ready for rain? A view from SPEC research on the future of cloud metrics[J]. arXiv preprint, arXiv: 1604.03470, 2016



Li Wei, born in 1999. Master candidate. His main research interest includes serverless computing.
李　威, 1999 年生. 硕士研究生. 主要研究方向为无服务器计算.



Li Guanghui, born in 1970. PhD, professor, PhD supervisor. Senior member of CCF. His main research interests include wireless sensor network, serverless computing, and intelligent nondestructive detection technology.
李光辉, 1970 年生. 博士, 教授, 博士生导师. CCF 高级会员. 主要研究方向为无线传感网、无服务器计算、智能无损检测技术.



Zhao Qinglin, born in 1974. PhD, professor, PhD supervisor. His main research interests include edge computing, privacy computing, blockchain and decentralized computing, and quantum computing.
赵庆林, 1974 年生. 博士, 教授, 博士生导师. 主要研究方向为边缘计算、隐私计算、区块链与去中心化计算、量子计算.



Dai Chenglong, born in 1992. Lecturer. His main research interests include electroencephalogram processing, electroencephalogram analyzing, and serverless computing.
代成龙, 1992 年生. 讲师. 主要研究方向为脑电图处理、脑电图分析、无服务器计算.



Chen Si, born in 1999. Master candidate. Her main research interest includes serverless edge computing.
陈　思, 1999 年生. 硕士研究生. 主要研究方向为无服务器边缘计算.