Adaptive and Efficient Resource Allocation in Cloud Datacenters Using Actor-Critic Deep einforcement Learning

Introduction

云计算的资源分配是指分配计算、存储和网络资源以满足用户。

目前资源分配的困难主要在以下几个方面：

- 数据中心的复杂性：需要提供计算资源和存储资源
- 用户需求的多样性：不同的任务需要不同的异构资源和不同的使用时长
- 能源消耗：能源消耗会造成巨大的运营开销，但是维持高能效的同时难以满足用户的需求。
- 动态云系统：云计算中资源的调度和状态经常变化，很难针对动态云系统构建准确的资源分配模型。

先前的云资源分配的解决方案有：基于规则、启发式和控制理论。这些方法通常利用了云系统的先验知识：状态的转换、需求变化和能源消耗等。

传统的强化学习在处理复杂的云环境时难以解决状态空间维度高的问题，可以使用DNN进行降维。

大多数方法使用的是基于值的DRL，在处理较大的动作空间时训练效率比较低，不符合云计算中心里作业不断到达的场景。

基于策略的DRL可以处理较大的动作空间，但是因为策略梯度产生的高方差会导致训练效率降低。

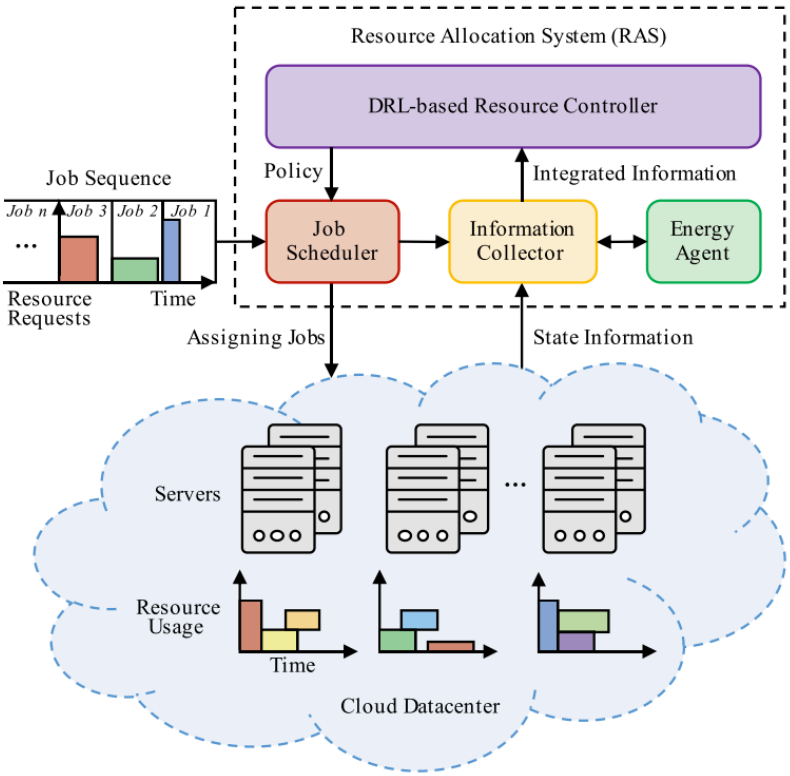
有人提出基于值和策略协同的A2C算法，但是采用单线程训练，计算资源利用率低，进而提出了A3C算法，使用多个DRL智能体同时交互，充分利用计算效率。

contributions

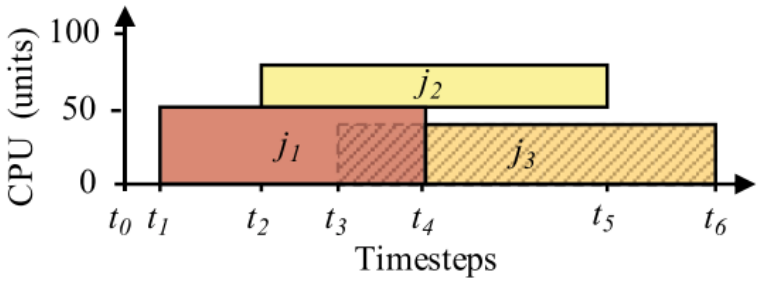
作者提出了一种基于A3C的资源分配方案，以QoS（作业延迟和失效率）和能源效率（平均作业能耗）作为优化目标，将动作空间和奖励函数公式化为马尔可夫决策过程。使用DNN处理数据中心的高维状态空间问题，使用多智能体DRL之间策略参数进行异步更新。

System Model

一个云计算系统的由m台服务器$V = \{v_1, v_2, \ldots, v_m\}$和每台服务器所拥有的资源$Res = \{r_1, r_2, \ldots, r_n\}$组成，$n$代表资源的种类。



设有jobs j_1, j_2和j_3，分别需要50，30和40的CPU资源，并且在t_1、t_2、t_3时间到达，预计持续时间为d_1, d_2, d_3，在t_4、t_5、t_6结束。



L_{normal}表示标准化平均作业延迟：执行任务j实际花费的时间/预计的时间

$$L_{normal} = \frac{\sum_{j \in completed\ jobs} \left(\left(T^j_{finish} - T^j_{enter} \right) / d_j \right)}{Number\ of\ completed\ jobs},$$

$disRate$表示任务序列满时，被取消的任务的比例：

$$disRate = 1 - \frac{Number\ of\ completed\ jobs}{Total\ number\ of\ jobs},$$

资源利用率上升时，可以打开更少的服务器提供更多的服务，服务器的能耗与其资源使用量成正比。云数据中心的总能耗可以表示为：

$$E_{total} = \sum_{t=0}^{T} \sum_{v \in V} \left(k \cdot P_{max} + (1-k) \cdot P_{max} \cdot U_t^{res} \right),$$

P_{max}表示一台服务器充分利用时的最大能耗，分数k表示一台空闲服务器的能量消耗比例，U_t^{res}表示t时刻的资源利用率。与现有的工作不同，作者考虑的是能量的效率：能量消耗与完成任务数量之比。

$$E_{job} = \frac{E_{total}}{Number\ of\ completed\ jobs}.$$

优化目标：提高QoS（降低L_{normal}和$disRate$），降低能量消耗（E_{job}）

RAS：Resource Allocation System资源分配系统

将RAS作为RL智能体，云数据中心视为环境

状态空间

状态空间S，t时刻的状态$s_t \in S$由所有服务器的资源使用情况和所有到达作业的资源请求组成。U_t^{res}代表t时刻所有服务器不同资源类型的使用情况，O_t^{res}代表t时刻每个任务的不同类型资源的占用，D_t^{job}表示t时刻作业持续的时间。云数据中心在t时刻的状态$s_t = [s_t^V, s_t^J] = [U_t^{res}, [O_t^{res}, D_t^{job}]]$

当作业完成时，状态发生变化，状态空间的维度计算$mn + z(n+1)$，m、n、z分别是服务器数量、资源类型和到达的工作数量

动作空间

作业调度器的动作a_t：根据策略在作业序列中选择执行作业。

策略是根据当前状态生成的，作业调度器将作业分配给特定的服务器执行，服务器会根据作业的资源请求自动分配资源。

动作空间定义为一个作业是否被服务器处理：$A = \{a_t | a_t \in \{0, 1, 2, \ldots, m\}\}$，$a_t = 0$代表不在t时刻分配作业，作业进入作业序列。

状态转移矩阵

状态转移概率的值是通过运行DRL算法根据当前的状态动态获得。

奖励函数

奖励包括QoS奖励和能量效率奖励

$$R_t = R_t^{QoS} + R_t^{energy}.$$

其中R_t^{QoS}代表各种资源延迟的惩罚，与时刻t的所有任务等待时间、执行时间和任务取消的时间有关。

$$R_t^{QoS} = -\sum_{j \in J_{seq}} \left(w_1 \cdot \frac{T_t^{j,wait} + T_t^{j,work}}{d_j} + w_2 \cdot T_t^{j,miss} \right),$$

R_t^{energy}代表t时刻的能源消耗惩罚，与t时刻所有任务消耗的能量有关。

$$R_t^{energy} = -w_3 \cdot \sum_{j \in J_{seq}} E_t^{j,exec},$$

Algorithm

基于A3C的资源分配方案。

主要流程：

1. 根据当前云数据中心的状态选择调度动作
2. 计算奖励函数、计算长期折扣回报，获取下一步的环境s_{t+1}
3. 计算critic的优势函数：$Q_w(s_t, a_t) = R_{disc} + \lambda^t V^{\pi_{\theta_t}}(s_{t+1})$:
$A^{\pi_{\theta_t}}(s_t, a_t) = Q_w(s_t, a_t) - V^{\pi_{\theta_t}}(s_t)$;
4. 最小化时序差分的误差：$\delta^{\pi_{\theta_t}} = R_t + \beta V^{\pi_{\theta_t}}(s_{t+1}) - V^{\pi_{\theta_t}}(s_t)$;
5. 更新状态-动作值函数的参数：$w_{t+1} \leftarrow w_t + \gamma_c \delta_t^{\pi_{\theta_t}} \nabla_w Q_w(s_t, a_t)$;
6. 更新actor的策略梯度：$\nabla_{\theta_t} J(\theta_t) = E_{\pi_{\theta_t}}[\nabla_{\theta_t} log\pi_{\theta_t}(s_t, a_t) A^{\pi_{\theta_t}}(s_t, a_t)]$;
7. 更新调度策略和状态：$\theta_{t+1} \leftarrow \theta_t + \gamma_a \nabla_{\theta_t} J(\theta_t); s_t = s_{t+1}$

策略梯度函数定义为：$\nabla_{\theta_t} J(\theta_t) = E_{\pi_{\theta_t}}[\nabla_{\theta_t} log\pi_{\theta_t}(s_t, a_t) Q^{\pi_{\theta_t}}(s_t, a_t)].$

采用时序差分学习可以准确估计状态值，在估计梯度时使用状态值函数降低方差，

根据当前的奖励值和动作值函数更新actor网络和critic网络的梯度

Algorithm 2. The Asynchronous Update of Policy Parameters of Job Scheduling in Each DRL Agent

Initialize: The global and local parameters (θ and θ') for actor's networks, the global and local parameters (w and w') for critic's networks.
for $i = temp - u, temp - u + 1, \ldots, temp$ **do**
  Accumulate gradients in the actor:
  $d\theta \leftarrow d\theta + \nabla_{\theta'} log\pi_{\theta'}(s_i, a_i)(R_i - V_w(s_i))$;
  Accumulate gradients in the critic:
  $dw \leftarrow dw + \partial(R_i - V_w(s_i))^2/\partial w'$;
end
Update global parameters by using gradient ascent via RMSProp: $\theta = \theta + \gamma_a d\theta,\ w = w + \gamma_c dw$;
Synchronize local parameters: $\theta' = \theta,\ w' = w$;
Reset gradients: $d\theta \leftarrow 0,\ dw \leftarrow 0$;

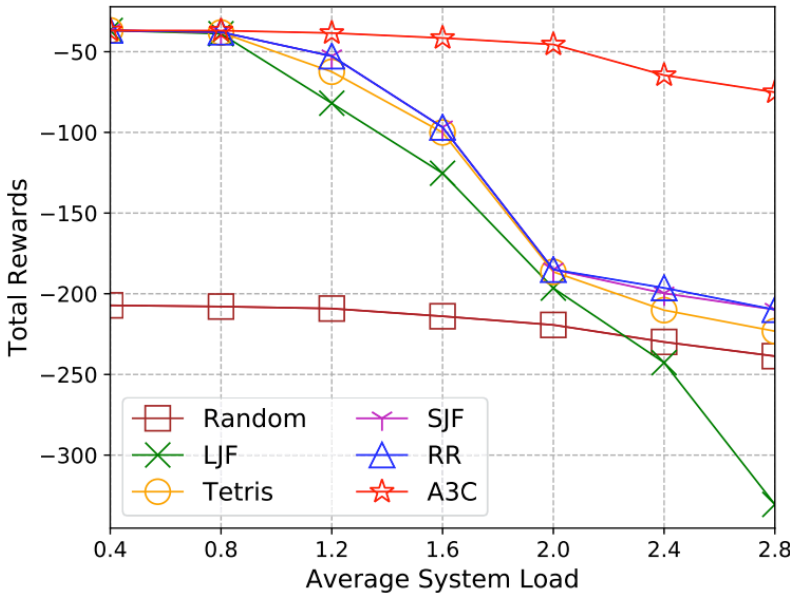
Evaluation

对标算法

- 资源分配算法
  - 随机任务执行(Random)
  - 最长任务优先(LJF)
  - 最短任务优先(SJF)
  - 时间片
  - Tetris
- 训练算法
  - PG
  - DQ

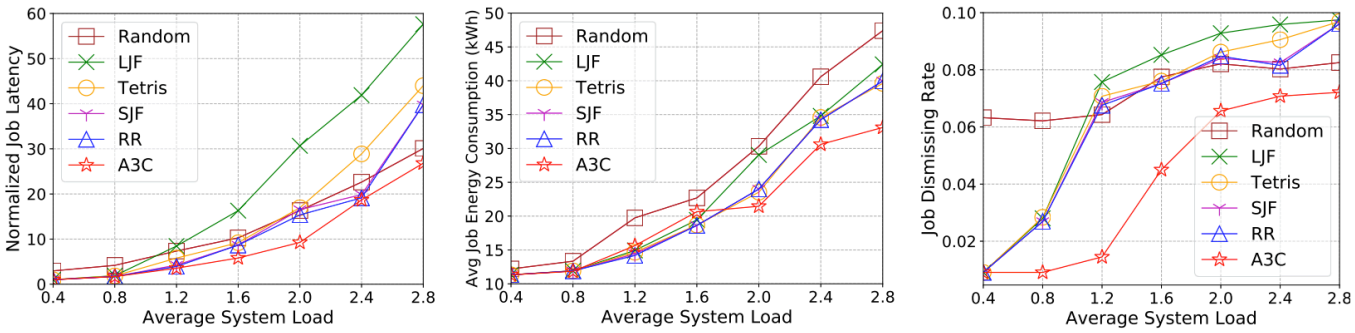
指标对比

- 单目标总回报(QoS)



相同负载下A3C能保持更好的总回报。最长任务优先因为优先满足最长的任务，会造成队列等待时间变长。

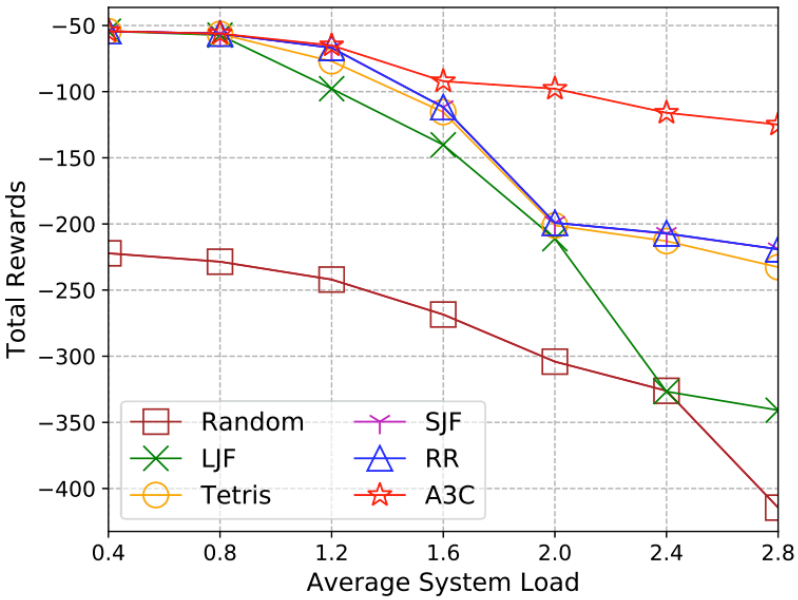
- 任务延迟、能量消耗和任务失败率

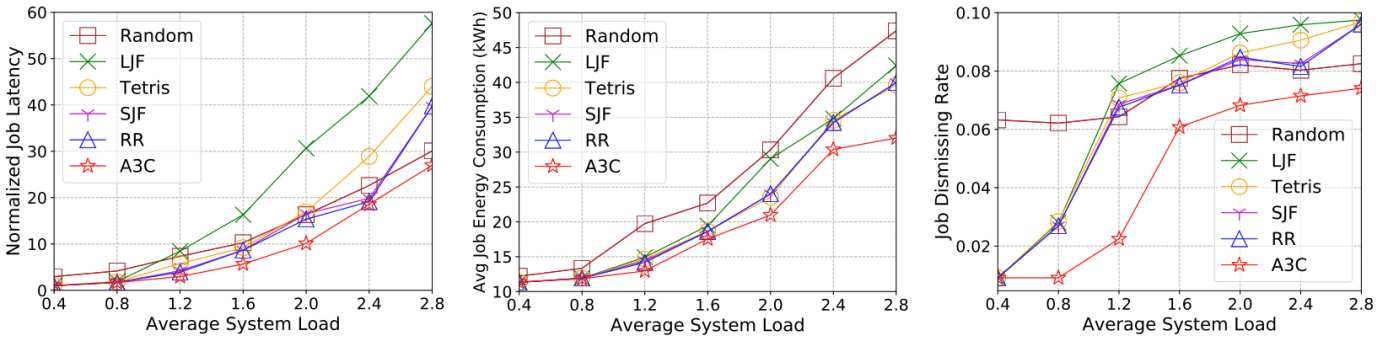


(a) Normalized job latency. (b) Average energy consumption of jobs. (c) Job dismissing rate.

随着平均系统负载变大，任务失败率的性能差距越来越大，证明A3C算法在系统负载动态变化时具有强适应性。

- 多目标总回报(QoS+能量消耗)



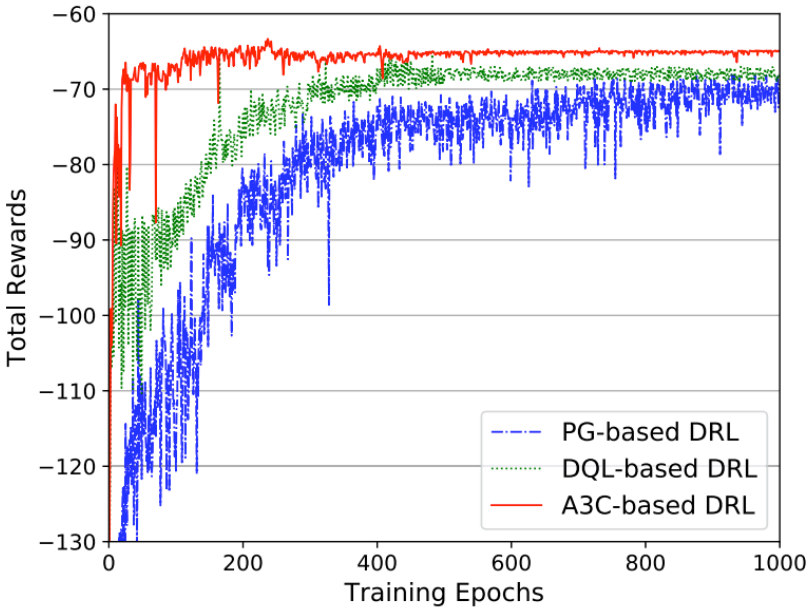


(a) Normalized job latency. (b) Average energy consumption of jobs. (c) Job dismissing rate.

- 训练速度对比

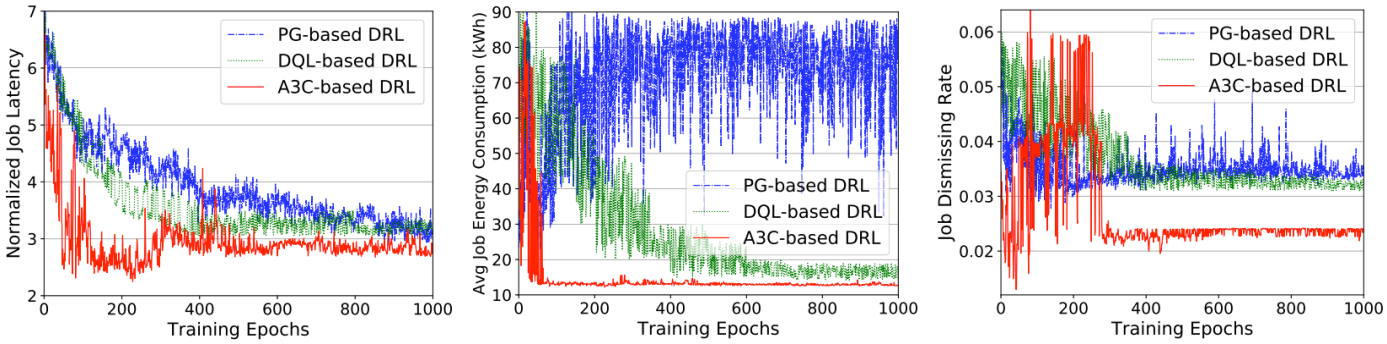
将A3C与PG和DQL进行对比，采用系统平均负载=1.2。

A3C的收敛速度、平均回报均比其他两个算法更好



PG因为在估计策略梯度的时候会产生高方差，因此无法实现良好的负载均衡，导致服务器高负载低利用率。

A3C能够同时实现低延迟和低能耗



(a) Normalized job latency. (b) Average energy consumption of jobs. (c) Job dismissing rate.

Conclusion

通过实验证明，基于A3C的资源分配方案在QoS和能效方面优于经典的资源分配方案。训练效率方面也能更快收敛。

优点：建模的思路很清晰，实验充分

不足：没有考虑到作业的优先级、作业Deadline