National College of Ireland

Configuring serverless functions using Q learning and Deep Q learning algorithms

MSc Research Project
MSc Cloud Computing

Johns Thomas
Student ID: 22203389

School of Computing
National College of Ireland

Supervisor: Sai Emani

National College of Ireland

MSc Project Submission Sheet

School of Computing

Student Name:	JOHNS THOMAS
Student ID:	22203389
Programme:	MSc Cloud Computing **Year:** 2024
Module:	Research Project
Lecturer:	Sai Emani
Submission Due Date:	12th August 2024
Project Title:	Configuring serverless functions using Q learning and Deep Q learning algorithms
Word Count:	8371 **Page Count:** 21

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project.
ALL internet material must be referenced in the bibliography section. Students are required to use the Referencing Standard specified in the report template. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action.

Signature:

Date: 12th August 2024

PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST

Attach a completed copy of this sheet to each project (including multiple copies)	□
Attach a Moodle submission receipt of the online project submission, to each project (including multiple copies).	□
You must ensure that you retain a HARD COPY of the project, both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer.	□

Assignments that are submitted to the Programme Coordinator Office must be placed into the assignment box located outside the office.

Office Use Only	
Signature:	
Date:	
Penalty Applied (if applicable):	

Configuring serverless functions using Q learning and Deep Q learning algorithms

Johns Thomas
x22203389

Abstract

Serverless functions attains great attention in recent years particularly because of its advantages like low administrative overhead, automatic scaling and fined grained control over billing. Developers can easily deploy service to cloud environments using serverless functions within seconds and are provided with few configuration options. Configuration like memory and timeout in commercial platforms like AWS Lambda directly affects performance and cost. Therefore, it is crucial to configure serverless functions with optimal parameters. Serverless function configuration becomes easy if the underlying relationship between configurations and cost is known. This research is an attempt to study the relationship between performance and configuration of a serverless function using reinforcement learning techniques such as Q learning and Deep Q learning. The Q learning and Deep Q learning agents have been developed and trained to learn the optimal configuration on serverless functions. This includes defining state and action space, reward function, and collecting the execution details of function execution after each invocation. Image processing functions deployed on AWS Lambda platform is used as the environment for the agents to interact with. Both Q learning and Deep Q learning agents have provided positive results in learning the relationship between performance and configuration of serverless function. The analysis of results show that Deep Q learning has edge in learning the relationship compared with Q learning. However, both agents can improve the performance by increasing state space by considering parameters like concurrency.

1 Introduction

Serverless Functions are the modern way of delivering software services without explicitly hosting a server and reducing the mundane tasks associated with managing and configuring backend servers. Large enterprises heavily invest in cloud automation and a growing trend is observed in the adoption of serverless architecture. As per Verified Market Research[1] (2024), the serverless architecture market size valued 12.3 billion USD in 2023 and is expected to reach 42.4 billion USD by 2031. Autoscaling, reduction in maintenance and pay-per-use pricing lurks businesses to serverless architectures. Developers do not require to provision virtual machines or create containers for serverless functions. Programmers only must develop the business logic and set some configurations. The cloud providers are responsible

[1] https://www.verifiedmarketresearch.com/product/serverless-architecture-market/.

for managing and maintaining servers and charge the users for each second of execution time of serverless function.

Although serverless computing reduces deployment time and maintenance; by eliminating the need for physical resource management and maintenance, configuration of memory and other parameters plays pivotal role in performance. Serverless platforms like AWS Lambda, Apache OpenWhisk explicitly provide option to configure memory, while others like Azure Functions and Google Cloud Functions hides it from users. The best configuration of memory and parameters like timeouts, concurrency of a serverless function plays a pivotal role in meeting cost, performance, and delay constraints. Earlier approaches to improve the serverless environments focussed on autoscaling, cold start latency reduction, function scheduling and so on. Only few studies such as Akhtar et al. (2020) and Wen et al. (2022) tried to model the relationship between configuration and performance (and cost).

In this research, image processing is taken as an area of interest. The proliferation of image-intensive applications has driven a surge in demand for efficient and cost-effective cloud-based image processing solutions. Serverless functions are utilized for tasks like image resizing, rotation, grayscale conversion, image format conversion, thumbnail creation, filtering, watermarking and so on. The configuration of serverless functions involving these tasks is taken to study the relationship between configuration and performance. Because of the dynamic nature of serverless functions, reinforcement learning (RL) has been chosen to study the relationship between configuration and performance in this research. So, the study tries to answer the following research question.

Along with the insights from literature survey conducted around the serverless environments gave rise to possibility of configuring serverless environments with reinforcement learning, formulates following research question.

Can reinforcement learning be used to model the relationship between performance (run-time, cost) and configurations of a serverless function to effectively configure resources?

By addressing above research question, the following objectives are aimed to be satisfied:

- Develop a reinforcement learning agent (in this case Q learning agent and Deep Q learning) capable of learning optimal serverless function configurations
- Integrate the trained RL agent with real world serverless platform like AWS lambda.
- Evaluate and compare the performance of the trained RL agents (Q learning & Deep Q learning agents) in terms of execution time, resource utilization, and cost-effectiveness compared to traditional static configuration approaches.

The project specifically focusses on image processing functions to avoid wide range of serverless functions deployed in different domains. It is not feasible to consider other applications in this scope of research. Also, memory is taken as the primary parameter which undergoes optimization during training. However other parameters like concurrency and cold start latency increase the state space of RL agent thereby increasing training time of the RL agent. Also, data for training the model is generated by on-the-fly invocations of deployed serverless functions during training, accounting for limited variety in the training dataset.

Overall, the report is structured as follows: Section 2 takes you through the earlier approaches in resource configuration and use of reinforcement learning in serverless environments. In section 3 detailed description of the methodology followed in the research is presented. Design specification and Implementation details of configuring serverless functions using

reinforcement learning is outlined in section 4 and 5. The outcome of study and results of training RL agent is deeply discussed in Section 6. It follows a conclusion and future work section to present further scope of research.

2 Related Works

Serverless functions have revolutionized cloud computing with enabling of abstraction of underlying infrastructure allowing developers to execute functions without explicitly managing servers. Although several cloud service providers offer serverless environments, optimising serverless function configurations remains a complex challenge.

2.1 Performance and cost optimisation in serverless functions

Serverless functions offer few configurations options for developers. Therefore, performance and cost optimisation in serverless functions is challenging and plenty of research were conducted.

Akhtar et al. (2020) explored statistical learning methods to predict the performance of different configurations in serverless functions. Their optimisation technique employed Bayesian optimisation to configure serverless functions. Bayesian optimisation is used to model the performance of functions, while integer linear programming is used to find configuration. The authors evaluated the effectiveness of the model through metrics such as performance, cost-efficiency and scalability. Additionally, the solution presented by Akhtar et al. (2020) is adaptable to several environments which strengthens their approach. However certain challenges remain unattended which includes model assumptions and complexity causing computational overhead. Although Bayesian optimisation is a powerful technique, it struggles with high-dimensional search spaces. Akhtar et al. (2020) sheds no light on the increasing number of configuration parameters and applicability of dimensionality reduction. The assumption of static workloads, especially in serverless context, weakens the model's reliability.

Autonomous Serverless Analytics with Cost-Efficiency and QoS-Awareness (ASTRA) proposed by Jarachanthan et al. (2021) designed to optimise memory configuration of map-reduce workloads running on serverless functions, specific to analytics jobs. After mathematically modelling the performance and cost of the workflow, the authors leveraged Dijkstra's algorithm to figure out optimal configuration, which turns out to be the shortest path in the workflow. The Directed Acyclic Graph formed consists of five layers, with each layer representing an aspect of optimisation problem. Nodes represent the memory configuration; edges represent the choice of certain configuration and weights denote completion time. The authors observed 60% improvement in performance when budget is fixed and 80% cost reduction without violating the SLOs. However, some assumptions by Jarachanthan et al. (2021), weakens their claims. For instance, it is assumed that all mapper and reducer functions have the same memory configuration. Like Safaryan et al., (2022), number of memory configurations increases the complexity of algorithms. Additionally, there are not enough evidence to claim the scalability and generalisability of the model using different type of mappers and reducers.

Apart from balancing cost and performance of serverless functions, studies were also conducted to consider service level objectives (SLO) while configuring the functions. Safaryan et al., (2022) introduced a tool 'SLAM (SLO-Aware Memory Optimisation)', that is

designed to optimise memory configurations of serverless functions based on service level objectives specified by the user as well as objectives like minimum cost or execution time. Distributed tracing is utilised by Safaryan et al., (2022), to model the execution time of different serverless functions at various memory configurations. The determination of optimal memory setting involves the consideration of the SLO requirements and user specified settings. The authors claim that service level objectives were met for 95% of serverless function configurations done using SLAM. The dependency of SLAM on tracing data makes it susceptible to inaccurateness or incompleteness in optimisation results. The adaptability to varying workloads also requires frequent re-optimisations. Unlike Akhtar et al. (2020) and Jarachanthan et al. (2021), SLAM focusses on multiple serverless functions.

Robert et al. (2020) introduced Serverless Application Analytics Framework (SAAF) to predict the performance and cost of serverless functions focussing on analytical jobs. SAAF enables profiling and characterisation of serverless functions performance, resource utilisation and infrastructure metrics. SAAF introduced by the authors aims in improving the accuracy of runtime prediction for serverless functions deployed with various configurations, mainly leveraged Linux CPU time accounting principles and regression modelling. Robert et al. (2020) assumed a homogenous workload, which did not actually reflect the variability in workload characteristics of production environment. Such assumptions impact the accuracy of performance and cost predictions. The simplification of workload characteristics significantly affects SAAF's ability to capture real world scenarios. Further exploration on the resource utilisation metrics and their impact on prediction can strengthen SAAF as the current selection of metrics and their thresholds for indicating prediction errors may not capture all relevant features.

Some attempts were made to model the performance and cost of serverless workflows as well. Lin & Khazaei (2021) investigated the possibility of predicting cost and response time of given serverless workflow's orchestration and configuration. Their solution includes formally defining a serverless workflow and analytical modelling to predict cost and response time. The authors tried to optimise cost and performance using Probability Refined Critical Path Greedy algorithm (PRCP). Lin & Khazaei (2021) depicted serverless workflow as weighted directed graph including elements like transition probabilities, delays, response time and costs. The PRCP algorithm enables handling of complex workflows like branches, cycles, parallelism and self-loops. Lin & Khazaei (2021) validated their claims thorough experimental validation and showed 98% accuracy for performance and 99% accuracy for cost estimation. Like Robert et al. (2020), the input size is assumed to be average for functions and fails to capture the dynamic workload characteristics, which affects the real-time applicability of the predictions and optimisations.

To calculate the abstract performance measure of resource scaling strategies in Faas platforms, Manner & Wirtz (2022) came up with a methodology. Their methodology provides a standard way to compare open-source Fass platforms with commercial serverless platforms in terms of performance and cost. The authors utilised Kubernetes and conducted performance tests using CPU-intensive functions. Manner & Wirtz (2022) tried to eliminate resource wastage and noisy neighbour problems by using Kubernetes limits and accounting for the difference in single-threaded and multithreaded functions. Although the authors presented a standardised way for performance comparison, they overlooked important factors like reliability, security, and scalability. The generalisability of their approach is limited due

to over-dependency with Kubernetes deployment for configuring open-source Faas platforms.

Kim aet al. (2020) proposed a fine-grained CPU cap control mechanism to solve performance degradation in serverless computing platforms. Their technique involves dynamically adjusting the CPU caps of collocated worker processes to reduce resource contention. Depending on the application groups and performance metrics like queue length and throttled time, the resource manager makes CPU cap adjustments, which allows fine-grained control over resources and adds efficiency. The solution put forward by the authors can handle workload variations and give automated management. While Kim aet al. (2020) aim to minimise resource contention and improve performance, some trade-offs in terms of resource allocation among different application groups affect the overall system balance.

The attempts focussed on modelling performance and cost of serverless functions, but the nature of serverless environments undergoes rapid changes. These include dynamic workloads, concurrency, network conditions, and so on. To adapt to such changes, a continuous learning environment is required. Reinforcement learning seems to be a good solution.

2.2 Reinforcement learning (RL) based solutions

Optimisation problems that use reinforcement learning based solutions in serverless functions mainly focussed on dynamic workloads and autoscaling, cold start latency reduction, function scheduling and resource configuration.

Zafeiropoulos et al. (2022) managed to clearly identify the challenges associated with autoscaling mechanisms in serverless computing platforms such as resource inefficiency. Their approach involved reinforcement learning to auto scale resources in serverless environments for adaptive resource management. Zafeiropoulos et al. (2022) experimented with different RL algorithms like Q-Learning, deep Q learning, and DynaQ+ and involved in discussion of metrics and reward function design. The author's approach differs from past efforts like static, rule-based autoscaling methods and outperforms them. The application of model free Rl algorithms makes it adaptable for dynamic workloads. However, the RL approach involves complex integration and considerable changes to the existing platforms which hinders the real-world applicability of the approach.

In their study on the use of RL for resource-based auto-scaling in serverless edge apps using OpenFaaS, Benedetti et al. (2022) investigated the use of the Q Learning algorithm to learn appropriate scaling policies and adjust the CPU usage threshold. The impact of different CPU utilisation thresholds on application latency was experimented by the authors using Kubernetes HPA. The results show that latency improves when CPU utilisation was set between 30% and 50%, with 30% yielding the best performance. However, Benedetti et al. (2022) only focussed on CPU utilisation and neglected memory and network usage. RL environment's state space definition is limited and leaves space for further development. Also testing the model on real world system opens the world to further improvements.

Bensalem et al. (2023) introduced reinforcement learning based solution to auto scaling problems in edge networks for efficient scaling and resource allocation of serverless

functions. Because their work focuses on edge networks, authors focussed on attaining lower average delays as compared to monitoring-based methods when high arrival rates and tight delay constraints are in place. Authors claim that RL and deep RL based approaches outperformed delay-aware monitoring approaches in performance. Bensalem et al. (2023) presented substantial evidence using simulations using 10 edge network nodes and different function types. Contrary to Zafeiropoulos et al. (2022) which involved a detailed exploration of memory and configuration space (discrete and continuous), Bensalem et al. (2023) focus on delay sensitivity and do not provide a detailed view of tuning and optimisation of reward function in RL and deep RL. Bensalem et al. (2023) open a promising direction and require additional validation and consideration of factors like reliability and robustness under varying network conditions.

Another notable work in the field of performance optimisation of serverless functions in edge networks using deep reinforcement learning was done by Yao et al. (2022). Their contribution involves an Experience-Sharing Deep Reinforcement Learning (ES-DRL) methodology which aims at enhancing efficiency of function offloading by combining serverless computing with edge networks. ES-DRL is more complicated when compared to Bensalem et al. (2023) where a distributed learning strategy and a population guided search method are introduced to accelerate convergence of RL agent to overcome local optima. ES-DRL mitigates issues associated with traditional DRL agents like insufficient sample diversity and high exploration cost. Additionally, Yao et al. (2022) conducted comparison studies with traditional offloading methods such as Greedy, and Random. However, the authors neglected the resource constraint nature of edge platforms where DRL based methods consume a lot of energy and complexity.

Somma et al. (2022) addressed management and scaling of containers in serverless computing environments. Their contribution includes the minimisation of resource contention and prediction of service times through core-restricted container provisioning. Using the cgroup feature in Linux, they suggested that allocating specific cpu cores to containers reduces resource contention, Like Zafeiropoulos et al. (2022), the authors used Q learning based autoscaling strategy and compared is efficiency with Kubernetes' Horizontal Pod Autoscaler (HPA). The authors claim that the system outperforms HPA in saving costs and predictable service times with low blocking rates. Somma et al. (2022) avoided hyper threading overheads entirely without trying to find the feasibility of it in applicable situations. Additionally, in cloud environments, CPU architecture varies significantly, but the assumption of a homogenous environment in terms of CPU architecture challenges their claims.

It is observed that studies of serverless function autoscaling in edge networks focussed on the reduction of response time and neglected memory configuration. Also, the complex nature of reinforcement learning, questions applicability in resource constraint environments.

Exploiting the ephemeral nature of serverless function execution, Suresh & Gandhi (2021) aimed to improve resource utilisation through collocating serverless functions with serverful applications (VMs). The author’s contribution includes the dynamic regulation of CPU, memory and last-level cache to ensure that colocation does not affect latency aware customers. Suresh & Gandhi (2021) reported that significant improvement in resource utilisation while maintaining performance degradation below 10% for serverful applications. The primary weakness in the author’s methodology is to ensure that colocation does not impact performance. Also, the methodology introduced by the authors requires precise

monitoring and regulation of resources which can be expensive and complex, which limits its applicability to only specific environments.

A predictive controller scheme was put forward by HoseinyFarahabady et al. (2018) to solve the issue of shared-resource contention in serverless platforms. They considered the interference among collocated Lambda functions when making resource allocation decisions as in Suresh & Gandhi (2021). The authors used a proactive approach by continuously monitoring shared resource capacity, interference among collocated functions, and resource utilisation at every host. The proactive approach taken by HoseinyFarahabady et al. (2018) helped to optimise resource allocation and minimise performance degradation in the Lambda platform. Additionally, cost functions are utilised to reduce the total QoS violation incidents and to keep cpu utilisation within the range. However, the predictive controller used closed loop system to monitor and adjust resource allocations which potentially adds complexity and overhead.

Cold- start time is an important part in serverless function optimisation around which a lot of research is conducted. Vahidinia et al. (2022) proposed a two-layer approach having reinforcement learning to discover function invocation patterns while using Long Short-Term Memory (LSTM) to predict future invocation patterns and number of pre-warm containers required. The authors tried to address the main criticism of existing solutions by predicting invocation patterns and keeping only the required number of pre-warmed containers instead of a fixed policy which led to memory wastage. The resource intensive training process and limited availability of dataset question the practicality and scalability of the method presented by Vahidinia et al. (2022).

Fifer is a resource management framework to tackle inefficiencies in serverless platforms particularly due to microservice agnostic scheduling and container over provisioning proposed by Gunasekaran et al. (2020). Fifer works in the context of function chaining where it is conscious of the container utilisation to scale containers based on the function characteristics and batches requests wisely. Moreover, to improve response time and adhere to service level objectives (SLO), fifer tries to avoid cold starts by proactively spawning containers. Gunasekaran et al. (2020) introduced the concept of slack, in which the difference between execution time and overall response latency was used to optimise batch size. Like Vahidinia et al. (2022), to deal with cold start fifer includes a LSTM based load prediction model. Fifer offers improved resource utilisation and energy savings up to 31%. The overhead associated in LSTM model and unpredictable workloads affect the performance of fifer.

Agarwal et al. (2021) presented a Q learning based solution to reduce cold start in serverless environments. In their study, the Q learning agent interacts with the environment to obtain per-instance CPU utilisation, available function instances, and success or failure rates of responses. The agent learns the workload patterns and adapts to environments such that it can determine optimal number of function instances ahead, to reduce cold starts. In comparison to Vahidinia et al., the dynamic nature of Q learning agent and ability to learn continuously make the strategy conceptualised by Agarwal et al. (2021) compatible to unknown invocation patterns. The large state-action space in the author's work associated with reinforcement learning takes larger training times and discretizing the cpu utilisation levels may contribute to suboptimal results.

Research was also conducted about scheduling functions in serverless environments. Pigeon and FnSched were two such approaches. Pigeon is an enhanced serverless framework for private cloud environments presented by Ling et al. (2019). Their study introduced a

function level resource scheduler on top of Kubernetes to handle the limitations of Kubernetes' native scheduling for short-lived functions. Pigeon improved resource utilisation as well as reduced cold start latency. The authors conducted an empirical evaluation using performance metrics, which showed 80% improvement in function cold start trigger rate, three times increase in throughput compared to AWS lambda and Kubernetes native scheduler-based serverless platforms. It is worth noting that there is a potential bias in the comparative analysis done by Ling et al. (2019). Their comparative study used specific versions of some frameworks. Any bias in the configuration of these frameworks impacts the results.

FnSched introduced by Gandhi & Suresh (2019) aims to minimise service provider costs with acceptable application latencies. The authors used single invoker scheduling to manage resource contention among different application containers by leveraging an application-aware CPU-shares regulation algorithm. FnSched also uses a greedy algorithm to avoid cold starts by reusing previously used invokers and autoscale and pack resources on fewer invokers. In their study, Gandhi & Suresh assumed that function execution times are fixed and can be estimated via profiling. Also, manual categorisation of functions is required and co-locating multiple functions may lead to performance degradation.

Reinforcement learning offers a promising path to learning the performance cost relationship thereby accurately configuring serverless functions. The exploration versus exploitation trade-off has the potential to figure out the optimal configuration. Notably, a gap exists in academia about configuring serverless functions with reinforcement learning. Most of the research typically involved autoscaling or reducing cold start time.

3 Research Methodology

The research aims to address the challenge of optimizing serverless functions configurations to optimize cost and runtime. The deductive approach using literature survey formed the basis for hypothesis, that is to utilize reinforcement learning (RL) to study and find the optimal configuration. The research methodology follows an experimental research strategy in optimizing configurations in serverless functions especially image processing tasks. The goal is to determine configurations for serverless functions that minimizes execution time, resource usage, and cost while maintaining acceptable performance. RL based method is promising compared to static rule-based approaches. RL based approaches ensures the dynamicity and adaptivity, since the learning process is continuous, and the established policies can be adjusted in reaction to changes in the cloud environment.

Furthermore, the methodology involves the development of a self-governed mechanism capable of finding the best configuration for serverless functions amid different workloads using reinforcement learning (RL), Q-learning and deep Q-Learning specifically. As for the RL problem, the task entails determining the best memory configuration and timeouts. The purpose is to reduce the cost of function execution time while satisfying the performance constraints.

3.1 State and Action Space Definition

3.1.1 Q-Learning

The reinforcement learning (RL) agent requires an environment to learn the relationship between cost and performance in serverless functions. Algorithms like Q-learning can learn the state-action pairs value (Q value) through interaction with the environment, without knowing the transition probabilities and reward function in advance. It is suitable for complex environments like serverless functions as modelling of such environment is highly complex. Q-Learning updates Q-values using the Bellman equation, based on the reward function and the estimated optimal future value, Sutton & Barto (1998).:

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left(\text{reward} + \gamma \max_{a'} Q(s',a') - Q(s,a)\right)$$

where, s is the current state, a is the action taken, s′ is the resulting state, and max over a′, Q(s′,a') is the maximum Q-value of the next state.

The RL environment can be defined using sate space, action space and reward function. The state is the representation of information available to RL agent at any point of time during interaction with serverless environment. In this context, state space includes parameters such as memory size, timeouts, and execution duration. Timeouts refers to maximum execution time allowed for serverless functions in seconds or minutes. The execution duration is discretised into intervals of 500ms in the range of [0, 15000]ms. The memory space is segmented into 128MB increments within the range [128, 3008]MB. The discretisation of the state parameters is necessary to tackle the complexity of the large continuous space. Each combination of these parameters refers to a particular state. Let's represent each state as tuple as given below,

State= (memory, timeout, duration categories)

Therefore, total number of states, S is the product of the number of possible values for each parameter:

S= |Memory sizes|×|Timeouts|×|Duration categories|

The action space defines the set action the RL agent can take for transition between states. In the serverless environments, the configurable parameters are memory size, and adjusting timeouts. RL agents can use discrete or continuous action space. However, to simplify the methodology and to gain a faster convergence, action space is chosen to be discrete. The possible options can be increment or decrement the parameter or retain the same. These options can be applied on memory size, and timeout. The possible action on memory is to increment or decrement the memory by 128MB or retain the same. Similarly for timeout, increase or decrease or retain same.

3.1.2 Deep Q learning

Deep Q learning (DQL) is an enhanced version of Q learning with deep neural networks (DNN). DQL helps to avoid complexity of maintaining large table having state and action combinations using deep neural networks, Zafeiropoulos et al. (2022). DQL have the capability to train the RL agent in large continuous space to find an approximated Q table. DQL helps to have a continuous state and discrete action space. The memory space is taken as continuous space in interval [128. 3008]MB. The execution duration and timeout are

retained as same as of Q learning. Otherwise, it would take considerably large amount of time to find a convergence.

3.2 Reward function Definition

The reward function is designed to incentivize the RL agent to find optimal configurations for serverless functions. The reward function determines the signal, RL agent receives for its actions. The reward function is composed of penalty for execution time, resource utilization, and cost. A negative reward proportional to the runtime of serverless function penalizes slow execution and encourages faster processing. To discourage overprovisioning of resources, negative reward proportional to utilized resources (memory) is taken as penalty. For cost optimization a negative reward proportional to the estimated cost as per the cloud platforms pricing is taken.
The three penalties are combining to obtain a negative reward function. The weighted sum of the execution time, resource utilization and cost give the reward function. The weights determine the priority of performance and lower cost. The negative reward structure incentivizes the RL agent to discover optimal configurations that balance cost and runtime effectively.

Reward = - ((Weight_Time * Execution Time) + (Weight_Resource * Resource Utilization) + (Weight_Cost * Cost))

3.3 Data Collection, Model Training and Evaluation

The training of RL agent involves interaction with real serverless environment like AWS Lambda. The optimal configurations for image processing functions that runs on serverless functions are to be determined by the RL agent. For Q-learning, the Q-table is initialized with zeros. The training process is divided into episodes and is defined as a complete run from initial state to terminal state. The terminal state is reached when further learning is unnecessary, or a predetermined number of steps are completed. Here, predetermined number of steps is taken to mark the terminal state. The learning parameters includes learning rate(α), discount factor (γ), and exploration rate (ϵ).

During training, the RL agent selects an action based on epsilon -greedy policy. In each step of the episode, to ensure both exploration and exploitation, a random action is selected with probability ϵ, or an action with highest Q-value is chosen from Q-table with probability 1- ϵ in case of Q learning. In deep Q learning, DQL model is used to find the next possible action. The selection of random action based on exploration rate (ϵ) allows exploration of state and action space, Zafeiropoulos et al. (2022). The selected action is executed on the AWS lambda by changing the configuration and invoking the task. Using AWS CloudWatch the metrics like runtime, maximum memory used, and cost are observed and retrieved. Based on the collected data, the reward is calculated, and Q-table is updated using the Q-learning update rule in case of Q learning. For Deep Q learning, neural network model training happens in batches. The procedure is repeated until the episode finishes. To favour exploitation over exploration, exploration rate ϵ can be decayed with the progress of the training.

The training of the RL agents is utilising serverless functions for the image processing tasks and publicly available image dataset. The collected metrics provide data for comparison with baseline approaches. The effectiveness of Q learning and Deep Q learning is analysed by plotting and comparing the collected metrics obtained during the training process. The analysis on the reward obtained per episode, memory configurations and actual memory used by serverless functions, execution duration and cost give valuable insights about using these techniques in real world settings.

4 Design Specification

4.1 Architecture

The system comprises of several core components that interact with each other to facilitate learning the runtime and cost relationships in serverless platforms. The core components include:

- Serverless environment
- The Q-Learning/DQN agent
- The state creation interface
- Logging and monitoring infrastructure

The serverless environment is used for execution the user defined image processing functions. The serverless environment acts as the playground for RL agents to explore. The Q-learning and DQN agent interact with serverless platform to explore and exploit platform to make decisions on configurations for image processing functions deployed on serverless platform. An object storage is used to store the input images for image processing tasks. Serverless environment provide option for capturing the execution metrics such as duration, memory usage, and errors. Monitoring infrastructure collects the execution metrics of the user defined functions run on the serverless platform. While monitoring mechanism collects details of serverless function execution, logging ensures that execution of RL agent training is properly collected and stored.

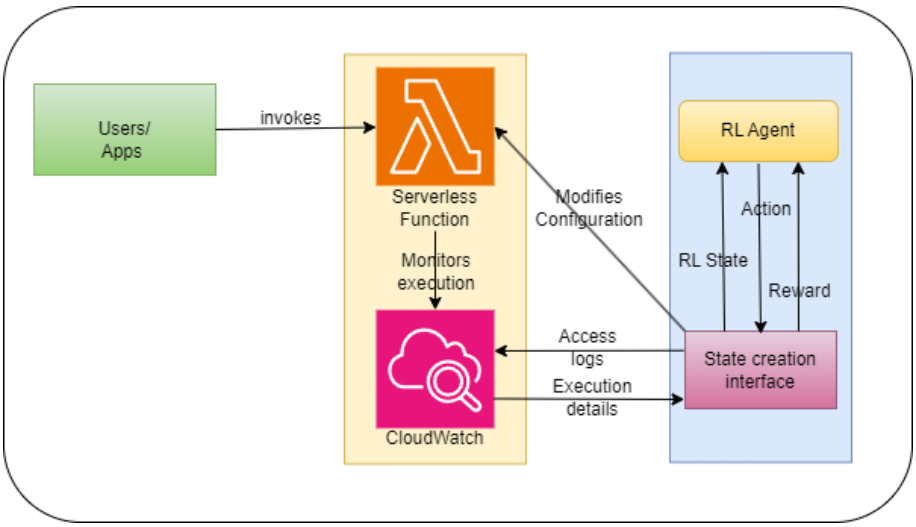

Figure 1: Overview of RL agent based Serverless function configuration

The collected metrics are analysed by state creation interface, which helps in determining the possible states and actions. The state in the context of serverless environment refers to

serverless function's configuration. The Q-Learning/DQL agent has state represented by memory, timeout and execution duration. The set of possible actions the agent can take includes increase memory, decrease memory, increase timeout, decrease timeout. Figure 1 shows an overview of proposal.

4.2 Q-Learning Agent

The Q-Learning algorithm starts with the initialization of Q-table with random values. Q-table is expected to store the rewards for each state-action pair. The state is represented as (memory, timeout, duration categories) as mentioned in section 3.1. The initial state of environment is set by configuring memory and timeout. Duration category is initialized to 0. The action that agent must perform is selected based on epsilon-greedy policy. The Q learning agent acts favouring exploration with probability ϵ. Also, with probability 1- ϵ, Q learning agent chooses action to support exploitation by taking the action with the highest Q-value for the current state.

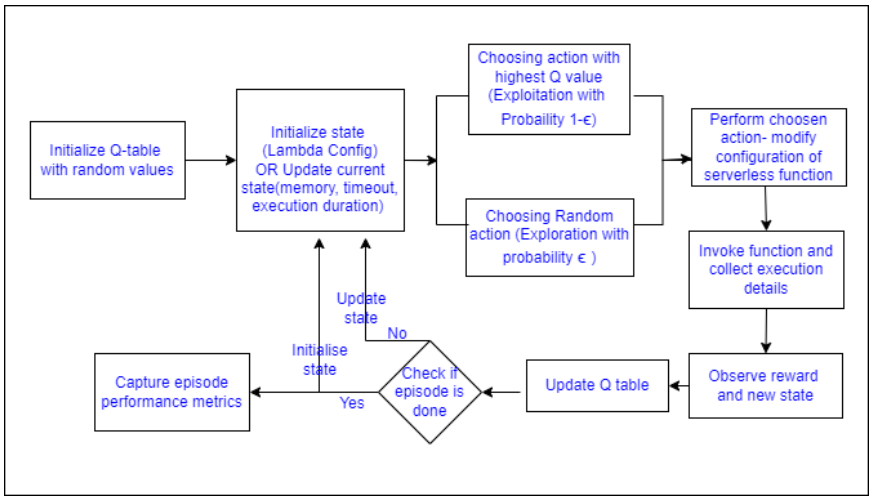

Figure 2: Q learning Agent Flow diagram

Based on the chosen action, serverless functions' memory and timeout is adjusted. The serverless function is invoked with new configurations. The performance metrics (execution time, memory usage, cost) of the function invocation are collected by monitoring infrastructure and reward is calculated using reward function. The Q-value for the current state action pair is updated using the Bellman equation. The current state is transitioned to a new state observed after function invocation, that is performed action. The transition to a new state marks the completion of a single step in the training episodes. Flow diagram of Q learning agent is given in Figure 2.

4.3 Deep Q-Learning (DQL/DQN) Agent

In comparison to Q learning, DQL agent uses a replay memory (buffer) to store experiences (state, action, reward, next state). Replay memory aids in breaking the correlation between consecutive experiences and improves stability of training, Mnih et al. (2015). The DQL agent uses a deep neural network(Q-Network) to approximate the Q-values for state-action pairs. Additionally, a target Q network can be used to provide stable targets during training, Zafeiropoulos et al. (2022). Q-Network is initialized with random weights and target Q network is also initialized with same weights as of Q-network. Flow diagram of DQL agent is given in Figure 3.

The state initialization happens by configuring memory and timeout. The action is selected based on the epsilon-greedy policy. With probability 1- ϵ, DQL agent chooses action to support exploitation by taking the action predicted by Q-network. The chosen action is executed by adjusting the memory and timeout configuration of serverless function. The serverless function is then invoked, and performance metrics is collected using monitoring infrastructure. The reward is calculated using observed metrics such as maximum memory used, duration, timeout errors and cost incurred. The transition to new state happens and current state, action, reward and next state are stored in replay memory. The experiences from replay buffer are sampled after predefined batch size is reached. The maximum Q-value of the next state predicted by the target Q-network and reward are used to compute target Q-value. The Q-network utilises predicted Q-values from the Q-network and the target Q-values from target Q network to calculate loss incurred during

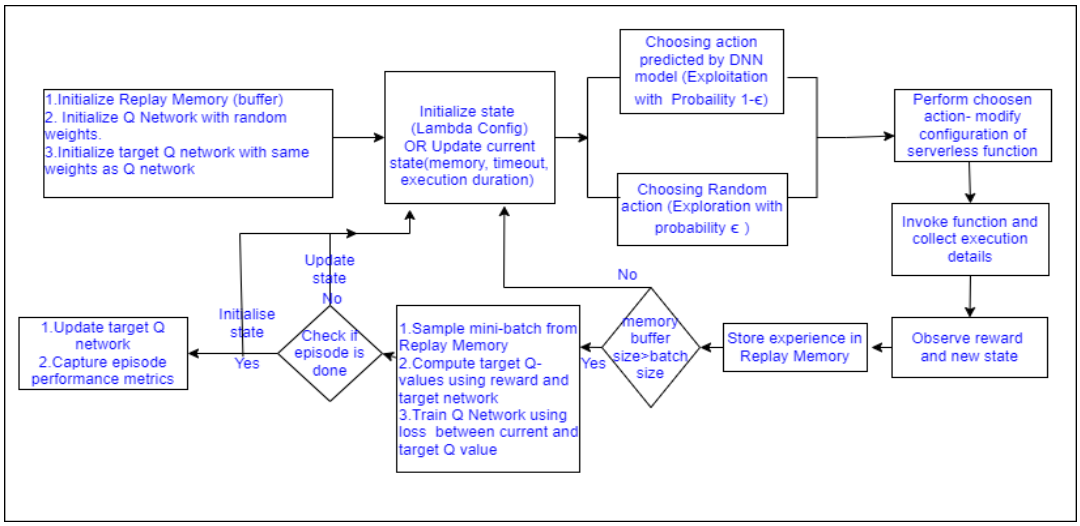

Figure 3: DQL Agent Flow diagram

training, Zafeiropoulos et al. (2022). The weights of the target Q-network are periodically updated to match the weights of the Q-network.

5 Implementation

The implementation involves defining the environment, Q learning and DQL agent, neural network for DQL agent and training of the developed agents. AWS lambda was chosen as the serverless environment for the reinforcement learning agent to interact with. The environment for Q learning and DQL agents involve AWS Lambda functions with configurable parameters: memory and timeout. As mentioned in Section 3, the memory space is segmented into 128MB segments within the range [128, 3008] MB for Q learning and defined as continuous space within range [128,3008] MB for DQL learning. The execution duration is discretised into intervals of 500ms in the range of [0, 15000] ms. Also, timeout is discretised into intervals of 2ms in the range of [0, 15] ms.

Python 3 programming language is used for the development of RL environment and Q Learning and DQL agents. Boto3 library is used to invoke lambda function and to retrieve log files from Cloudwatch. AWS CloudWatch is used to monitor the lambda function invocations to collect the function execution details. Image processing tasks like image

resizing, rotation, grayscale conversion, image format conversion, thumbnail creation, filtering and watermarking are used in training of Q Learning and DQL agents.

To facilitate the training of RL agents, each of the above-mentioned image processing tasks is deployed as a serverless function in AWS Lambda. For each of the deployed serverless functions, Cloudwatch was set up to monitor the function invocation and to collect execution logs. During each invocation of the serverless function invocation, an image from Flickr-Faces-HQ[2] (FFHQ) image dataset was used as the input for image processing task. The images from Flickr-Faces-HQ are stored in AWS S3 bucket. Flickr-Faces-HQ (FFHQ) dataset offers various images with different size and quality, which enables exploration of memory space during training.

The training process of the RL agents starts with initialisation of state space with memory=256MB and timeout =5ms duration=0ms. To promote exploration of state space, initial configurations for some episodes are randomly selected with probability ϵ.

Tensorflow library is used to develop the DQL model. The DQL neural network has an input layer, a hidden layer and output layer. The Mean Squared Error (MSE) loss function is used to measure the difference between the predicted Q-values and the target Q-values during the training of DQL neural network. The Adam optimizer is used to update the model weights based on the computed gradients. The summary of the Deep neural network used by DQL agent is given in Figure 4.

Model: "sequential"

Layer (type)	Output Shape	Param #
dense (Dense)	(None, 24)	96
dense_1 (Dense)	(None, 24)	600
dense_2 (Dense)	(None, 4)	100

Total params: 796 (3.11 KB)
Trainable params: 796 (3.11 KB)
Non-trainable params: 0 (0.00 B)

Figure 4: Summary of the Deep neural network used by DQL agent

Python script named qlearning_agent.py and dqn.py runs the training process for Q learning and DQL agents respectively. When training script is executed, the lambda functions are called with configurations as defined in each of the RL agents. To evaluate the efficiency of the training, the execution details collected using cloudwatch (max memory used, timeouts, duration) and are written to results.json file. The analysis of results.json provides the details of performance of Q learning and DQL agents.

6 Evaluation

6.1 Q-learning

The training of the Q learning agent done was over the course of 100 episodes. The training involved more than 1000 invocations of AWS lambda function, which allowed the Q-learning agent to learn the optimal policy during the training process. In each episode, the Q learning agent invokes the deployed lambda function and adjust the memory and timeout

[2] https://github.com/NVlabs/ffhq-dataset?tab=readme-ov-file#readme

based on the calculated reward. The training of Q learning agent took around 24 hours for 100 episodes on AMD Ryzen 7 machine. In real world, external applications and users invoke the lambda functions, Q learning agent only needs to work on the execution trace of the function execution.

Figure 5. shows the plot of average reward per episode. The reward remains negative across all episodes. The plot reveals that agent was not learning well up to 50 episodes. Later Q learning agent starts to learn the policy and try to attain stability. Also, the Q learning agent tries to avoid actions which results in very low rewards, as the average reward of most of the episodes is between -1 and -4 after episode 50. The variation in average rewards seems somewhat more contained in the latter episodes. The possible reason for the outliers in the plot may be due to out of memory error or timeout error.

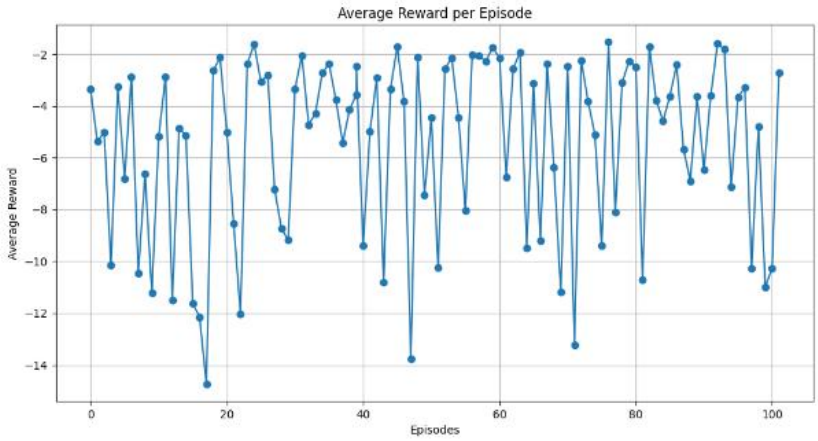

Figure 5: Q learning agent - Average rewards per Episode

The average cost incurred per episode during the training of Q learning agent can be inferred Figure 6. The Q learning agent tries to stabilize average cost incurred after 50th episode with little outliers.

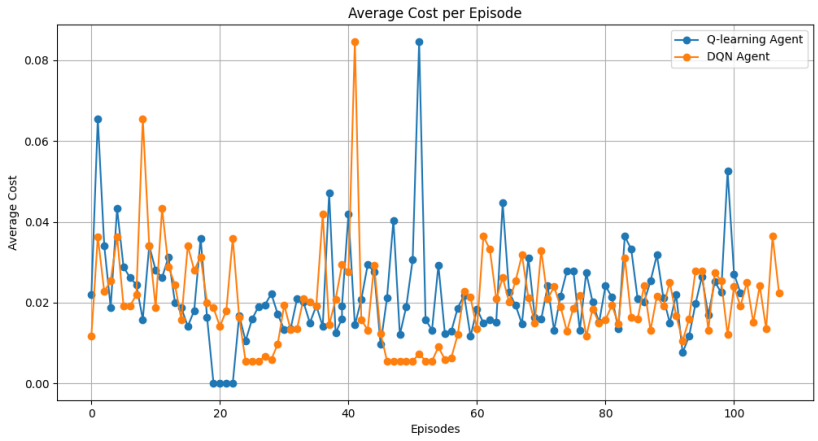

Figure 6: Q learning agent - Average cost per Episode

The average memory configuration per episode plot (Figure 7), reveals the reason for the outliers in the average reward per episode and average cost per episode plots. The agent is learning the optimal policy, so whenever the chosen action is not optimal it punishes the agent for its action. It is noted that with more number training episodes the q-learning agents can be improved and shows a promising path for exploration. From Figure 7, it is noticeable that the Q learning agent starts to learn the memory usage pattern of image processing tasks

on serverless functions after 60th episode. In serverless environments like lambda, memory configured also affects cpu time available. So, Q learning agent considered trade-off between performance and cost, which can be the reason for the difference in memory usage vs memory configured.

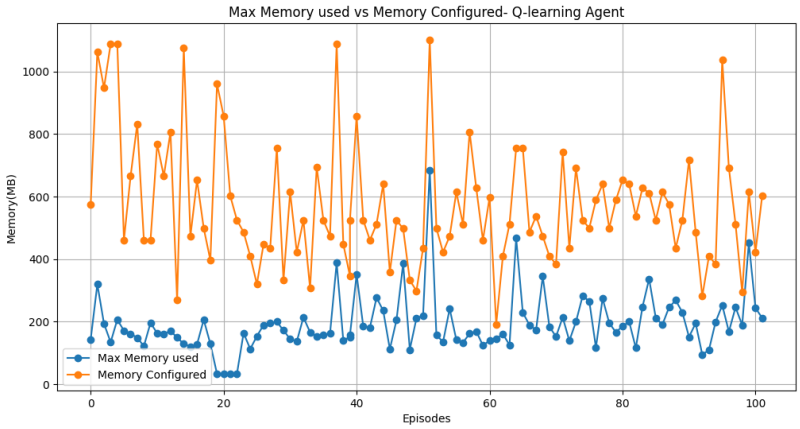

Figure 7: Q learning agent - Memory used vs Memory Configured

6.2 Deep Q learning Agent (DQN/DQL)

The Deep Q learning agent was trained over 100 episodes to learn the relationship between performance and configuration of serverless functions. Deep Q learning agent training involved 1000 invocations of image processing tasks deployed on the AWS lambda environment. The agent adjusts the memory and timeouts based on the previous experiences learnt by the deep neural network. The training of the DQL agent took around 28 hours, slowness of the training can be accounted due to the waiting time for retrieving the logs of function execution.

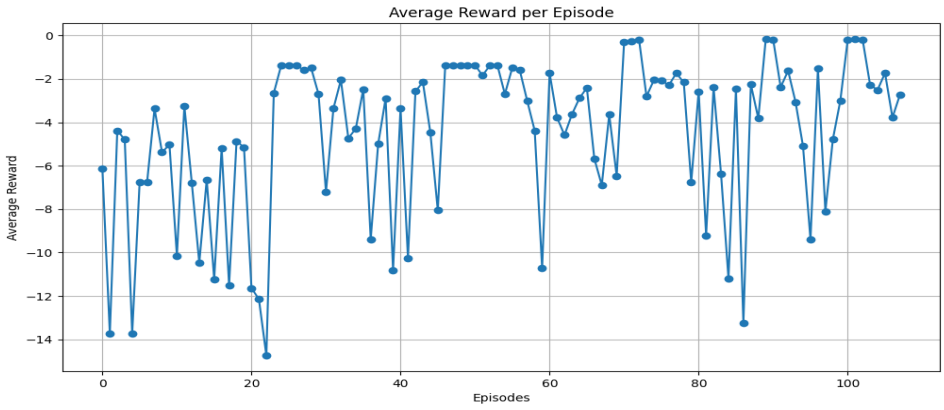

Figure 8: DQL Agent- Average Reward per Episode

DQL Agent performs comparatively better than Q-learning. Q learning agent was suffering from instability, as identified in Figure 5. While DQL agent tries to stabilize the reward between 0 and -4, although occasional outliers are present, refer to Figure 8. The presence of outliers may be due to exploration of state space by the DQL agent. The DQL agent was unstable during the start of training with out of memory errors and timeouts. After passing 40 episodes, the agent got better understanding of the serverless environment.

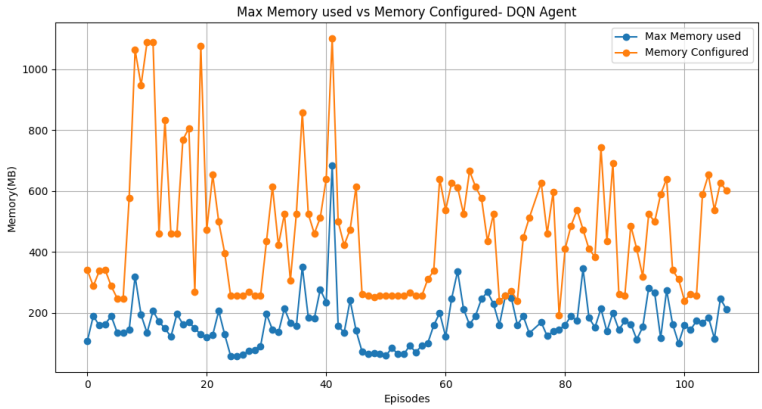

Figure 9. DQL Agent: Memory used vs Memory Configured

From figure 9, it can be inferred that DQL Agent started learning the pattern of memory usage of the functions and configures serverless functions accordingly after episode 42. Although there is a significant difference in configured memory and maximum memory used, DQL Agent started learning the pattern in memory usage of serverless functions. In serverless environments like lambda, memory configured also affects cpu time available. Therefore, DQL agent also gives importance to performance, which can be the reason for the difference in memory usage vs memory configured. Also from figure 6, it is evident that DQN agent shows cost optimization compared to Q-learning agent because DQN better learnt memory usage patterns. The plots of execution duration versus memory configured over 100 episodes for the Q learning and DQL agents are given in Figure 10. Plots in Figure 10 reveal that most of the image processing tasks discussed in the research falls under 400MB to 800MB memory configurations. Also, 400MB to 800MB memory configurations gives an execution duration of 3000 to 5000ms. Identifying such relations enable better optimising configurations in serverless environments.

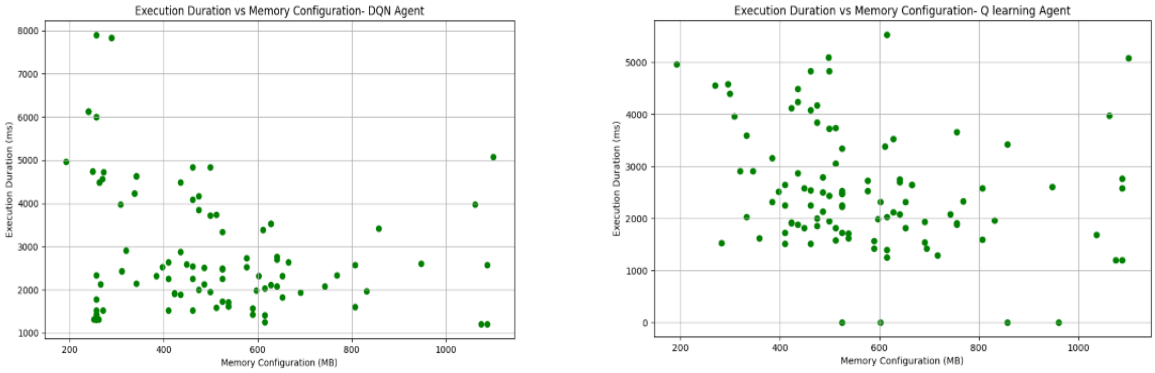

Figure 10: Execution Duration vs Average memory configured

The comparison with baseline approaches like manually configuring AWS lambda function with 128 MB (minimum assumed for image processing tasks discussed here) and 3008MB (maximum memory assumed in the research) shows that the Q learning and DQN learning agents succeeded in finding an optimal configuration. Execution Duration vs Average memory configured plot shown in Figure 10 implies that image processing serverless functions can be configured within 400MB to 800MB range to have optimal performance and cost.

6.3 Discussion

Reinforcement learning techniques, Q-learning and Deep Q learning can be effectively utilised to obtain optimal configuration for serverless functions. Deep Q learning outperforms Q learning in learning the relationship between configuration and performance. In this research the common problems like concurrency concerns and cold start problems are not considered for the training of the RL agents. It is because adding more parameters to state space increases the training time of the reinforcement learning agents as well as the time to converge to an optimal solution. During the training of RL agents, 100 episodes each having not less than 10 steps has been performed, but still have the scope of improvements. If number of episodes is increased, possibly 1000, RL agents may perform better. Q learning agent lacks stability compared to Deep Q learning agent in capturing the optimal configurations for serverless functions. Deep Q learning uses a continuous state space and neural network to capture the relations, which can be reason for its comparatively good performance.

In this research, serverless functions running on Python programming language is experimented. The research can be extended to functions using other language too. The image processing functions are cpu bound in nature. To validate generalisability of model in other domains experiments with I/O intensive and network intensive functions are required. Additionally, the results obtained over 100 episodes may be not seen as satisfactory from perspective of time taken for training. But, in real world systems with high invocation rates, with millions of requests, the idea of Q learning and Deep Q learning is promising and offer more advantages including cost reduction.

7 Conclusion and Future works

In this work, reinforcement learning techniques like Q learning and deep Q learning were utilized to study the relation between performance and configuration of serverless functions. The comparative analysis reveals that deep Q learning performed better than Q learning in finding optimal configurations. The serverless environment AWS lambda was chosen to interact with RL agents and execution logs were retrieved to provide feedback to RL agents. Image processing tasks were identified as the candidate serverless functions that undergoes configuration optimisation. The results show that both RL techniques possess good opportunity to further explore and experiment with serverless functions.

Q-learning and Deep Q learning agents can be integrated into real world serverless functions for optimising the configurations. Due to the active learning nature of these techniques, these RL techniques can better converge in serverless functions with high frequency invocations. In this research, certain image processing tasks are considered. However, these techniques can be expanded into other applications which are cpu intensive in nature. The I/O intensive and network intensive tasks may not benefit from the assistance provided in configuring parameters. Because configurations like memory determine the cpu allocation in the serverless functions. Further research options include using actor-critic reinforcement learning and implementing the solution with open source serverless platforms like Apache Openwhisk. Also increasing the state space dimensions can be considered. Moreover, choosing functions written in other programming languages like Java and node.js can be further explored.

References

Agarwal, S., Rodriguez, M. A. and Buyya, R., "A Reinforcement Learning Approach to Reduce Serverless Function Cold Start Frequency," *2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, Melbourne, Australia, 2021, pp. 797-803, doi: 10.1109/CCGrid51090.2021.00097.

Akhtar, N., Raza, A., Ishakian, V. & Matta, I. (2020), Cose: Configuring serverless functions using statistical learning, in '*IEEE INFOCOM 2020- IEEE Conference on Computer Communications'*, pp. 129–138.

Amoghavarsha Suresh and Anshul Gandhi. 2021. 'ServerMore: Opportunistic Execution of Serverless Functions in the Cloud'. *In Proceedings of the ACM Symposium on Cloud Computing (SoCC '21). Association for Computing Machinery, New York, NY, USA*, 570–584. https://doi.org/10.1145/3472883.3486979

Anshul Gandhi and Amoghvarsha Suresh. 2019. FnSched: An Efficient Scheduler for Serverless Functions. In Proceedings of the 5th International Workshop on Serverless Computing (WOSC '19). Association for Computing Machinery, New York, NY, USA, 19–24. https://doi.org/10.1145/3366623.3368136

AWS Lambda (2024), 'AWS Lambda', https://aws.amazon.com/lambda/. Accessed: July 15, 2024.

Azure Functions (2024), 'Azure Functions', https://azure.microsoft.com/en-us/services/functions/. Accessed: July 17, 2024.

Benedetti, P., Femminella, M., Reali G., and Steenhaut, K., "Reinforcement Learning Applicability for Resource-Based Auto-scaling in Serverless Edge Applications," *2022 IEEE International Conference on Pervasive Computing and Communications Workshops and other Affiliated Events (PerCom Workshops)*, Pisa, Italy, 2022, pp. 674-679, doi: 10.1109/PerComWorkshops53856.2022.9767437

Bensalem, M., Ipek, E. and Jukan A., "Scaling Serverless Functions in Edge Networks: A Reinforcement Learning Approach," *GLOBECOM 2023 - 2023 IEEE Global Communications Conference*, Kuala Lumpur, Malaysia, 2023, pp. 1777-1782, doi: 10.1109/GLOBECOM54140.2023.10437794.

Flickr-Faces-HQ Dataset (FFHQ), https://github.com/NVlabs/ffhq-dataset?tab=readme-ov-file#readme Accessed: July 17, 2024.

Gunasekaran, J.R. *et al*, 2020. "Fifer: Tackling Resource Underutilization in the Serverless Era". *In Proceedings of the 21st International Middleware Conference (Middleware '20)*. Association for Computing Machinery, New York, NY, USA, 280–295. https://doi.org/10.1145/3423211.3425683

HoseinyFarahabady, M. R., Zomaya, A. Y. and Tari, Z., "A Model Predictive Controller for Managing QoS Enforcements and Microarchitecture-Level Interferences in a Lambda Platform," in *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 7, pp. 1442-1455, 1 July 2018, doi: 10.1109/TPDS.2017.2779502.

Jarachanthan J., Chen L., Xu F. and Li B., "Astra: Autonomous Serverless Analytics with Cost-Efficiency and QoS-Awareness," *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, Portland, OR, USA, 2021, pp. 756-765, doi: 10.1109/IPDPS49936.2021.00085.

Kim, Y. K., HoseinyFarahabady, M. R., Lee, Y. C., and Zomaya, A. Y., "Automated Fine-Grained CPU Cap Control in Serverless Computing Platform," in *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 10, pp. 2289-2301, 1 Oct. 2020, doi: 10.1109/TPDS.2020.2989771.

Lin, C. & Khazaei, H. (2021), 'Modeling and optimization of performance and cost of serverless applic ations', *IEEE Transactions on Parallel and Distributed Systems* 32(3), 615–632.

Ling, W., Ma, L., Tian, C., and Hu, Z. "Pigeon: A Dynamic and Efficient Serverless and FaaS Framework for Private Cloud," *2019 International Conference on Computational Science and Computational Intelligence (CSCI)*, Las Vegas, NV, USA, 2019, pp. 1416-1421, doi: 10.1109/CSCI49370.2019.00265.

Manner J. and Wirtz G., "Resource Scaling Strategies for Open-Source FaaS Platforms compared to Commercial Cloud Offerings," *2022 IEEE 15th International Conference on Cloud Computing (CLOUD)*, Barcelona, Spain, 2022, pp. 40-48, doi: 10.1109/CLOUD55607.2022.00020.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A.A., Veness, J., Bellemare, M.G., Graves, A., M. Riedmiller, A.K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, D. Hassabis, Human-level control through deep reinforcement learning, Nature 518 (7540) (2015) 529–533, http://dx.doi.org/10.1038/nature14236.

Robert C., Hanfei Y., Varik H., Zohreh S., David F., David P., Rashad H., and Wes L., 2021. The Serverless Application Analytics Framework: Enabling Design Trade-off Evaluation for Serverless Software. *In Proceedings of the 2020 Sixth International Workshop on Serverless Computing (WoSC '20). Association for Computing Machinery, New York, NY, USA*, 67–72. https://doi.org/10.1145/3429880.3430103

Safaryan, G., Jindal, A., Chadha, M., and Gerndt M., *"SLAM: SLO-Aware Memory Optimization for Serverless Applications," in 2022 IEEE 15th International Conference on Cloud Computing (CLOUD)*, Barcelona, Spain, 2022 pp. 30-39. doi: 10.1109/CLOUD55607.2022.00019

Somma, G., Ayimba, C., Casari, P., Romano S. P., and Mancuso V., "When Less is More: Core-Restricted Container Provisioning for Serverless Computing," *IEEE INFOCOM 2020 - IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, Toronto, ON, Canada, 2020, pp. 1153-1159, doi: 10.1109/INFOCOMWKSHPS50562.2020.9162876.

Sutton, R. S. & Barto, A. G. (1998), Reinforcement Learning An Introduction, The MIT Press.

Vahidinia, P., Farahani, B., and Aliee, F. S., "Mitigating Cold Start Problem in Serverless Computing: A Reinforcement Learning Approach," in *IEEE Internet of Things Journal*, vol. 10, no. 5, pp. 3917-3927, 1 March1, 2023, doi: 10.1109/JIOT.2022.3165127.

Verified Market Research, “Serverless architecture market size, share, opportunities & forecast.” https://www.verifiedmarketresearch.com/product/serverless-architecture-market/. Accessed on July 20, 2024.

Yao, X., Chen, N., Yuan, X., and Ou, P., “Performance Optimization in Serverless Edge Computing Environment using DRL-Based Function Offloading," *2022 IEEE 25th International Conference on Computer Supported Cooperative Work in Design (CSCWD)*, Hangzhou, China, 2022, pp. 1390-1395, doi: 10.1109/CSCWD54268.2022.9776166.

Zafeiropoulos, A., Fotopoulou, E., Filinis, N. & Papavassiliou, S. (2022), ‘Reinforcement learning-assisted autoscaling mechanisms for serverless computing platforms’, Simulation Modelling Practice and Theory 116, 102461. URL: https://www.sciencedirect.com/science/article/pii/S1569190X21001507