San Jose State University
SJSU ScholarWorks

Master's Projects | Master's Theses and Graduate Research

Fall 2022

Cloud Provisioning and Management with Deep Reinforcement Learning

Alexandru Tol
San Jose State University

Follow this and additional works at: https://scholarworks.sjsu.edu/etd_projects

 Part of the Artificial Intelligence and Robotics Commons, and the Databases and Information Systems Commons

Recommended Citation

Tol, Alexandru, "Cloud Provisioning and Management with Deep Reinforcement Learning" (2022). *Master's Projects*. 1106.
https://scholarworks.sjsu.edu/etd_projects/1106

This Master's Project is brought to you for free and open access by the Master's Theses and Graduate Research at SJSU ScholarWorks. It has been accepted for inclusion in Master's Projects by an authorized administrator of SJSU ScholarWorks. For more information, please contact scholarworks@sjsu.edu.

Cloud Provisioning and Management with Deep Reinforcement Learning

A Project

Presented to

The Faculty of the Department of Computer Science

San José State University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

by

Alexandru Tol

December 2022

© 2022

Alexandru Tol

ALL RIGHTS RESERVED

The Designated Project Committee Approves the Project Titled

Cloud Provisioning and Management with Deep Reinforcement Learning

by

Alexandru Tol

APPROVED FOR THE DEPARTMENT OF COMPUTER SCIENCE

SAN JOSÉ STATE UNIVERSITY

December 2022

Dr. Fabio Di Troia	Department of Computer Science
Dr. Thomas Austin	Department of Computer Science
Dr. Robert Chun	Department of Computer Science

ABSTRACT

Cloud Provisioning and Management with Deep Reinforcement Learning

by Alexandru Tol

The first web applications appeared in the early nineteen nineties. These applications were entirely hosted in house by companies that developed them. In the mid 2000s the concept of a digital cloud was introduced by the then CEO of google Eric Schmidt. Now in the current day most companies will at least partially host their applications on proprietary servers hosted at data-centers or commercial clouds like Amazon Web Services (AWS) or Heroku.

This arrangement seems like a straight forward win-win for both parties, the customer gets rid of the hassle of maintaining a live server for their applications and the cloud gets the customer's business. However running a cloud or data-center can get expensive. A large amount of electricity is used to power the blades that inhabit the racks of the data-center as well as the air-conditioning that prevents those blades and their human operators from overheating. Further complications are added if a customer hosts in multiple locations. Where should incoming jobs be allocated too? What is the minimum number of machines that can run at each location that ensures profitability and customer satisfaction.

The goal of this paper is to answer those questions using deep reinforcement learning. A collection of DRL agents will take data from an artificial cloud environment and decide where to send tasks as well as how to provision resources. These agents will fall under two categories: task scheduling and resource provisioning.We will compare the results of each of these agents and record how they work with each other. This paper will show the feasibility of implementing DRL solutions for task scheduling and resource provisioning problems.

ACKNOWLEDGMENTS

I want to thank my girlfriend for putting up with me working on this project late at night every night. I also want to thank my family for encouraging me to pursue an education further than a bachelor's degree. Finally I'd like to thank Professor Di Troia for meeting with me frequently and giving me advice.

TABLE OF CONTENTS

1

Introduction

1.1 Datacenters and Clouds

A Cloud is a network of computing resources that can be accessed on demand from other computers/devices remotely. Clouds offer a way of hosting applications which is simpler and cheaper than setting up a physical machine in-house and a solution to storing large volumes of data. Clouds also serve as a means to outsource heavy computational work such as AI training. The term 'cloud computing' began gaining traction in the late 1990s and broke through to mainstream usage in 2006 when Eric Schmidt (the at the time CEO of Google) used the term in an industry conference [1]. However the concept of a centralized pool of computing resources accessed by users on remote terminals has been around since the 1960s with the usage of mainframe computers. Truly remote networked clouds became possible with client/server paradigm enabled by the UNIX OS and the SAP R/3 software [2].

Clouds offer many advantages to both personal and corporate users alike. Most users whether they be individuals or corporate only use about 10 percent of their CPU resources and 60 percent of their memory resources [3]. This means that most of the time, at least half of each user's computing resources are idle. It makes more economical sense for these users to pool their computing resources together and use only what they need at a fraction of the cost of maintaining their own in-house resources. This is where cloud service providers such as Amazon Web Services and Heroku step in. These companies provide computing resources and services such as storage,load balancing, and security. Clouds are able to provide these services through virtualization where the a single physical machine can have it's computing resources (CPU and memory) allocated separately among multiple users. The artificial cloud environment developed in this project will be based off of this concept.

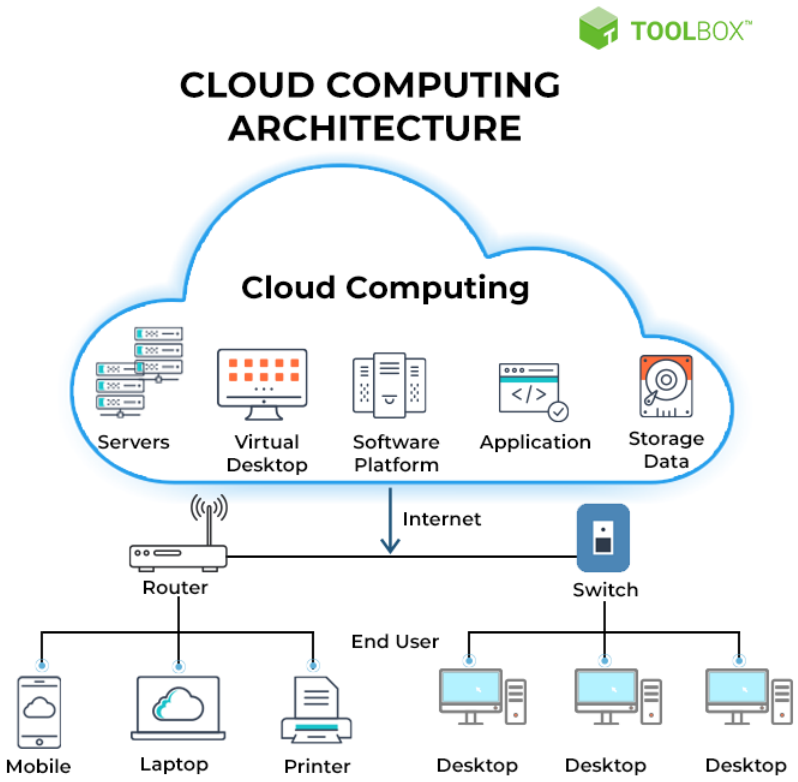

Figure 1: Standard topography of a cloud [4]

1.2 Deep Learning

Deep learning is a subset of machine learning used to solve problems from many different disciplines. Deep learning models have been employed in a range of problems such as but not limited to : natural language processing, fraud detection, autonomous driving, and medical diagnosing. Unlike machine learning which uses singular statistical models to make predictions on outcomes or classifications, deep learning models are composed of many of these statistical models in the form of neurons, the basic unit of computation for a deep learning model [5]. Neurons are organized into layers which are usually fully connected with each other, meaning that each node of a layer 'n' has a connection to each node of the next layer 'n+1'. There

are three kinds of layers : input,hidden, and output. The input layer is the layer that takes in either raw or reprocessed data and passes it on to the hidden layers. The output layer is what determines the final classification/response to the data, it takes the processed info from the hidden layers and assigns a probability value to each neuron (representing a classification or action). Finally the hidden layer is where the processing of the input data takes place, depending on the application deep learning networks usually have between 1 to 5 hidden layers.

Deep learning has a variety of advantages which make it an attractive tool to apply to a learn-able problem. One advantage is the flexibility that deep learning offers, it can be used in both supervised and unsupervised learning scenarios [6]. Another major advantage is that deep learning eliminates the need for data processing. While pre-processed data can still be passed to a deep learning model, the model will eventually learn which dimensions are useful and which are not. Also worth mentioning is that deep learning is a prolifically researched field, this means that there is a large selection of coding frameworks and research materials freely available for any developers to use. One of the most popular of these frameworks, Pytorch [7], is used in this thesis.

1.3 Solveable

As previously mentioned, cloud environment offer a variety of advantage at a fraction of the cost of traditional hosting methods, however the situation is not a simple win-win for both the customer and the cloud service provider. Since the cloud service provider is offering computing resources at a cost lower than that of an in-house setup, this means that their profit margins can be tight. This means that all computing resources must be used as efficiently as possible throughout the day, this means directing workloads to the closest available server/data-center and shutting off

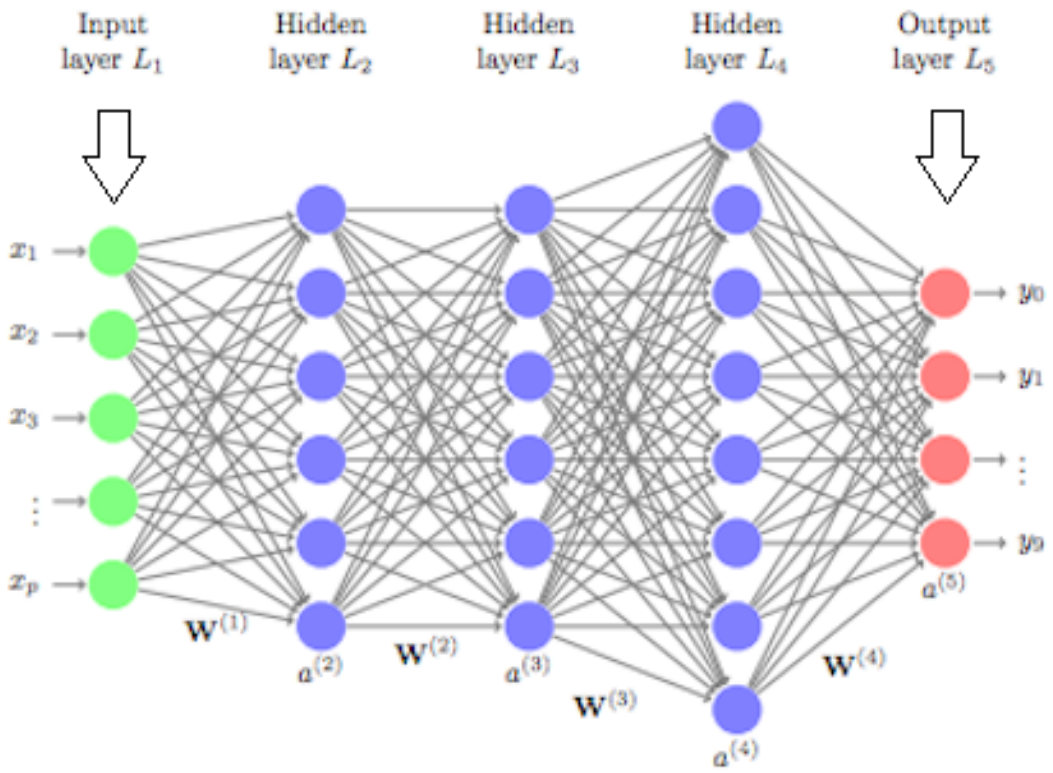

Figure 2: An example of a nueral network [8]

machines that are idle.

However there are many considerations that come at play, such as where to direct workloads. If a company hosts their application at a single data-center and get all incoming workloads from the region the data-center is located in, then directing the workloads is an easy task. However, if a company hosts their application in multiple regions and gets incoming requests from regions across the globe, then the task becomes less straight forward. Furthermore it's not immediately obvious how many computing resources should stay on at one time. Some companies will have heavy workloads occurring during normal works hours while others will have a more chaotic schedule.

Algorithms with hard coded limits can be applied to these problems but they can

only handle simple scenarios. Such algorithms are too rigid for cloud environment that's always changing. For task scheduling and resource provisioning problems deep reinforcement learning (DRL) is a perfect fit because DRL agents can learn from an environment in real time. In addition since this problem is a decision task and not a classification task, no labeling is needed.

2

Background

2.1 Reinfrocement Learning

Reinforcement learning is a training technique centered around a goal oriented algorithm such as a reward function. Classifications and decisions are made based off of values given by a reward function. Unlike more conventional programs the agent is not told what actions to take, instead it learns which actions are optimal by mapping reward values to the actions taken [9]. There are 4 basic elements to a reinforcement learning agent : a policy, a reward function, a value function, and an environment [9].

A policy is the value mapping between states and actions, it dictates what actions the agent should take for a given state. For example, a model that decides what a person should have depending on the weather can have a policy such that on hot days ice cream will have the highest values and on cold days the highest values will be for soup. The agent will use the policy to choose the action with the highest values given a state, so this means ice cream will be recommended on hot days and soup on cold days.

The reward function is function that assigns a reward based on an action 'a' taken at state 's' in context of the short-term. A value function is similar to a reward function except that it bases its reward on the long term. For example, the reward function will give a high value for ice cream on hot days. However a value function will assign a lesser value for ice cream because while its a good treat for hot days, having ice cream too many days in a row will increase your blood sugar and lead to negative health related side effects.

The final element, the environment, is what reacts to the agent's actions and gives feedback. An environment is not always needed since the agent can just get feedback from the reward and value functions. However an environment offers an

organic means to train a reinforcement learning agent that could lead to more a more practical policy.

2.1.1 Q-Learning Algorithm

Q-learning is a specific reinforcement learning algorithm used in unsupervised contexts. The Q learning algorithm tries to find the action for a given state that yields the highest reward by using its value function. The value function is updated after every action by using the Bellman equation. The Bellman equation updates the value function by trying to find an optimal sequence of actions. It does by finding the 'Q' value of every state action pair, for a given state action pair it finds the immediate reward for choosing action "a" at state "s" and adds it to the best possible future reward of another action taken at state "s'" [10].

$$Q(s,a) = r(s,a) + \gamma \max_{a} Q(s',a)$$

Figure 3: The Bellman Equation [11]

2.1.2 Deep Reinforcement learning

Deep reinforcement learning is the marriage between the main feature of deep learning (the neural net),the action-state-reward paradigm of reinforcement learning and the Q-learning algorithm. In deep reinforcement learning the policy is represented by the weights between each connected node, these weights will change as the agent learns. Studies have shown that deep reinforcement algorithms usually outperform conventional reinforcement algorithms [12].

2.2 Sequence to sequence learning

In most deep learning scenarios the input data is of a static size, e.g images fed to a classification agent will always have the same resolution. In most cases this arrangement is acceptable but what if the data is irregular like the resource

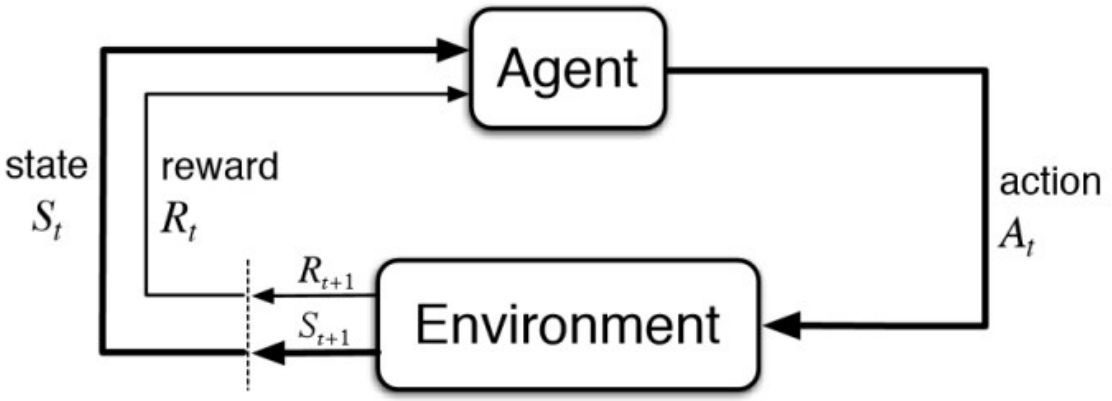

Figure 4: Standard Reinforcement Learning setup [13]

dimensions of a data-center's server? Servers will be powered on and shutdown per the data-center's needs so the size of a provisioning agent's input will be variable. The solution is to integrate a sequence to sequence framework.

Sequence to sequence agents are used in a variety of applications, most prominently natural language processing [14]. This is because sequence to sequence models are able to process variably sized input which makes them suitable for real world applications. While these agents offer an effective way to process real world data, certain challenges can arise when deciding on how many samples to process per run. For example, when training a scheduling agent to send out tasks, it makes more sense for the agent to process all available servers before making a decision instead of just processing one.

A typical sequence to sequence agent will be comprised of an encoder and a decoder. The encoder processes the incoming data and feeds it to the decoder which then produces an output [14]. Despite the total size of the data being variable, the encoder is able to process the data because the data itself is made up of 'n' regularly sized packets, a fact that will be very useful to this project.

3

Related Works

3.1 Overview

A variety of research papers and technical sources were used as inspiration and research material for this project. The works listed below are the ones most important to this project.

3.2 A deep learning-based resource usage prediction model for resource provisioning in an autonomic cloud computing environment

The paper "A deep learning-based resource usage prediction model for resource provisioning in an autonomic cloud computing environment" by Al-Asaly et al. uses a diffusion convolutional neural network to predict future CPU loads to an environment [15]. The learning agent's task was to use past CPU usage metrics to predict future CPU usage metrics, the past CPU metrics were supplemented by actual usage metrics from a live app. Specifically the paper wanted the agent to predict the future usages in the frames of 5,15,30 and 60 minutes. The agent saw modest results when trying to predict for these ranges but saw improved performance when implementing a deep-belief network. When the number of VMs in the computing environment were high, the agent would tend to make more accurate predictions [15].

This paper was a massive inspiration through the use of its 'automatic framework' derived from IBM's MAPE model. MAPE stands for : Monitor , Analyzer, Planner, Executor. The monitor observes the traffic comming into the computing environment and sends the data to the analyzer which then predicts what the number of future requests will be. The planner looks at the analyzer's predictions and then plans the best course of action. Finally the executor runs the commands specified by the planner. This setup was the main inspiration for the Executor, Observer, Orchestrator environment implemented by this project.

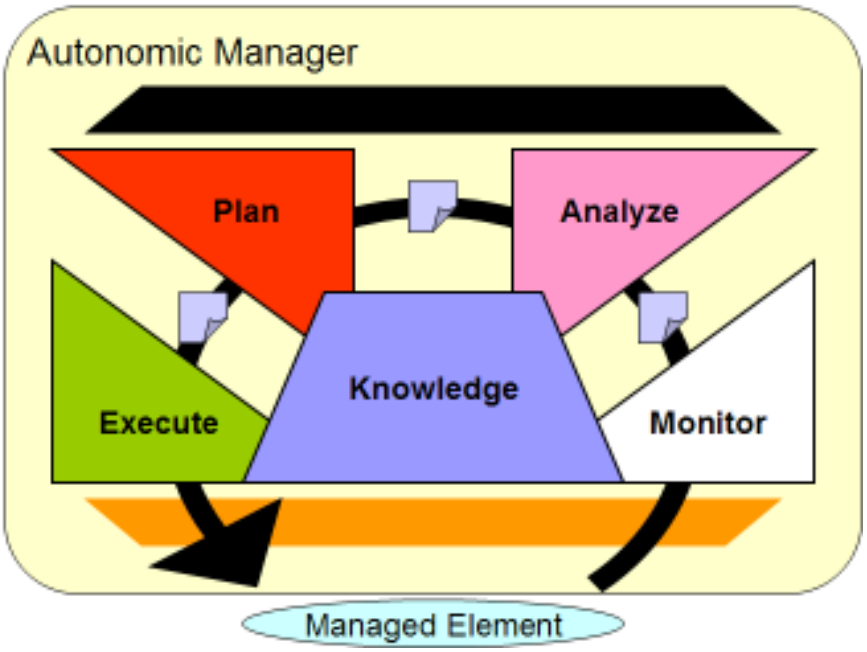

Figure 5: IBM's mape model [16]

3.3 An agent based model for resource provisioning and task scheduling using drl

This paper by Oudaa et al. [17] uses a deep learning agent to reduce the power consumption costs of a cloud/datacenter. Power consumption is calculated by adding static electricity consumption with dynamic electricity consumption. Static electricity consumption is due to a virtual machine (surrogate for a server) being turned on while dynamic electricity consumption is due to the server processing tasks which results in CPU usage. In addition the agent must work within a certain energy consumption threshold. If that threshold is not reached for a certain time period, no penalty is applied. If the threshold is breached then a cost penalty is applied.

Tasks are part of workloads which are Directed Acyclic Graphs (DAG) which simulate task dependencies (meaning that one task cannot be executed until another is complete). Two deep learning agents are tasked with sending tasks to their appropriate destinations. The first agent decided which server (VM) farm to send the task to,

the second decides which individual server (VM) in that farm should process the tasks. Each task has a reward payment associated with it which allows the agents to minimize the electricity cost by maximizing the task reward. The agent setup was compared to a traditional round robin setup and performed significantly better. In addition, the agent setup was shown to continuously improve over time, something a round-robin algorithm could never do [17].

This paper inspired two aspects of this project. The first was having a multi-agent setup which helped simplify the approach to handling all the actions required of a cloud environment. Training a learning agent to both schedule tasks and provision resources would have been extremely difficult and prohibitive. The second aspect that was taken as inspiration from this paper was the idea of generating tasks. This project implements task generation by combining that feature with the orchestrator which itself was inspired by the previous paper. The workloads as DAGs was something that was desired for this project but could not be implemented due to time constraints.

3.4 AI-Based Resource Provisioning of IoE Services in 6G: A Deep Reinforcement Learning Approach

In this paper the authors identify deficiencies in current methods employed by cloud service providers to predict incoming demand. To fill the gap in capability the researchers propose an AI driven solution by pairing a deep reinforcement agent with a heuristic algorithm. In this paper the researchers setup a cloud environment that is subject to a quality of service agreement (QOS). The QOS ensures that the customer receives a minimum level a service (either in the form of response times or availability), when violated this incurs a cost on the cloud provider. The job of the deep learning agent is to scale/provision resources in such a way that costs are minimized while quality of service is maintained [18].

There are four objectives the agent needs to achieve : minimize the application

load,minimize overload of available resources, minimize containers priority cost, minimize cost of other objectives. These objectives result in 4 cost functions that are further modified by scalars from the discount index. The optimum action can be seen as the minimum of all costs relating to a certain state (t-1) after action a was taken for a given future state (t). This gives us a Q-function that chooses the optimal choice for all future states [18].

The deep learning agent and heuristic algorithm combo yielded successful results and had the interesting effect of favoring some of the 4 objectives over others. This paper inspired the dimensions that would be used when training the provisioning agent, namely CPU and memory load.

3.5 Dynamic Input for Deep Reinforcement Learning in Autonomous Driving

In the paper by Huegle et al. [19], the researchers train an a deep learning agent in control of a car in a 2D car driving game (known as SUMO). In this paper, an emphasis is put on variable length input as that is how data is often encountered in real life situations. The researcher's answer to the question of varialble length input is the Deepset-Q algorithm. A deepset-Q agent involves 3 seperate agents and splits data between dynamic data and static data. The first agent processes the dynamic data which is split into n vectors of equal size. The second agent/module processes the data from the first agent to be fed into the last agent. Finally the last agent takes in the static data as well as the processed data from the second agent and makes a decision.

The deep set agent(s) outperformed all other agents including the "set2set" agent which the fromer was based on. While it's performance dropped as the number of cars in the scenario increased, the deep set agent(s) experieneced less loss than all the other agents [19].

Huegle et al.'s paper was instrumental in making this project possible by offering its solution to the problem of dynamic input by splitting that input into n equally sized vectors. It also inspired how schedule net would be structured into three separate agents. This setup resulted in more effecient learning since there are three agents learning to do simple tasks instead of one agent learning to do a complex task. While schedule net's agents have significantly different jobs than the agents of deep-set, it draws much inspiration from the latter agent.

3.6 Combined use of coral reefs optimization and multi-agent deep Q-network for energy-aware resource provisioning in cloud data centers using DVFS technique

Similar to Oudaa et al [17], the authors of this paper seek to reduce the power consumption of a computing environment. What is unique about this paper however is how the Asghari et al. [20] pair an optimization algorithm known as the "coral reef optimization algorithm" with a deep learning agent. The coral reef algorithm is used for short term decision while the deep learning agent is used for long term decisions.

The coral reef algorithm itself is based off of how corals reproduce in nature. The algorithm works as follows. A reef is initialized, where are reef is an array of corals. Each reef contains corals which represent a decision policy for assigning tasks. The corals are placed in their own "cells" within the reef. If a coral is within a cell by itself, there is no competition or mating. If a coral shares the cell with another, it can compete with it or mate. Mating involves creating a new coral that has features from both parent corals. After certain intervals, the corals with the worst performance are killed off while the more successful corals increase in numbers [20].

This paper showed an interesting way that deep learning agents can be paired with optimization algorithms and also introduced the concept of a markov game which mediates between the two entities so that their decisions don't collide.

3.7 Autonomous Maintenance in IOT Networks via AoI-Driven Deep Reinforcement Learning Algorithms

The paper by Stamatakis et al. [21] applies deep learning methods to problem of maintaining devices in IOT networks.The agent can only take 3 actions: no maintenance, network maintenance, sensor maintenance. AOI (Age of information) is used as the basis for reward function. The reward function at any time t is equal to 1 divided by the sum of the maintenance cost associated with an action plus the average AOI. The maintenance cost is 0 for no action taken. The learning agent was able to reduce the total number of network faults in the environment.

This paper showed another practical way deep learning agents can be applied to a cloud environment and gave further insight into how deep reinforcement agents work.

4

DataSet and implementation

4.1 Overview

The project consists of 3 main parts: the artificial cloud environment, the redis db, and the learning agents. The cloud environment will host the virtual machines (the stand in for data-centers) and within those virtual machines will be containers (the stand in for live servers). The artificial cloud environment will communicate directly with the learning agents using the TCP protocol. As requests are made and containers are provisioned, each action will be stored within the redis db. The progress of the learning agents will be tracked by writing the output of each respective reward function to a file.

4.2 Dataset

One novel aspect of this project will be the data-set used to train the task scheduling and provisioning agents. Instead of downloading a massive set of labeled/unlabled data, this project will instead generate its own data by using an artificial cloud environment. This environment will be made up of virtual machines (that represent datacenters) and containers (that represent servers). This environment will be able to create and delete containers as well as send tasks at the request of the learning agents.

4.3 The Cloud Environment

The cloud environment will consist of two main components: the controlling processes and the deployable entities (i.e: the virtual machines and containers). There are three controlling processes : the orchestrator, the observer, and the executor. These three processes are responsible for provisioning and keeping track of the containers and VMs as well as sending data to the learning agents.

This method was chosen because it offers an organic way for the learning agents to interact with the data and because it would be cheaper than actually using a live

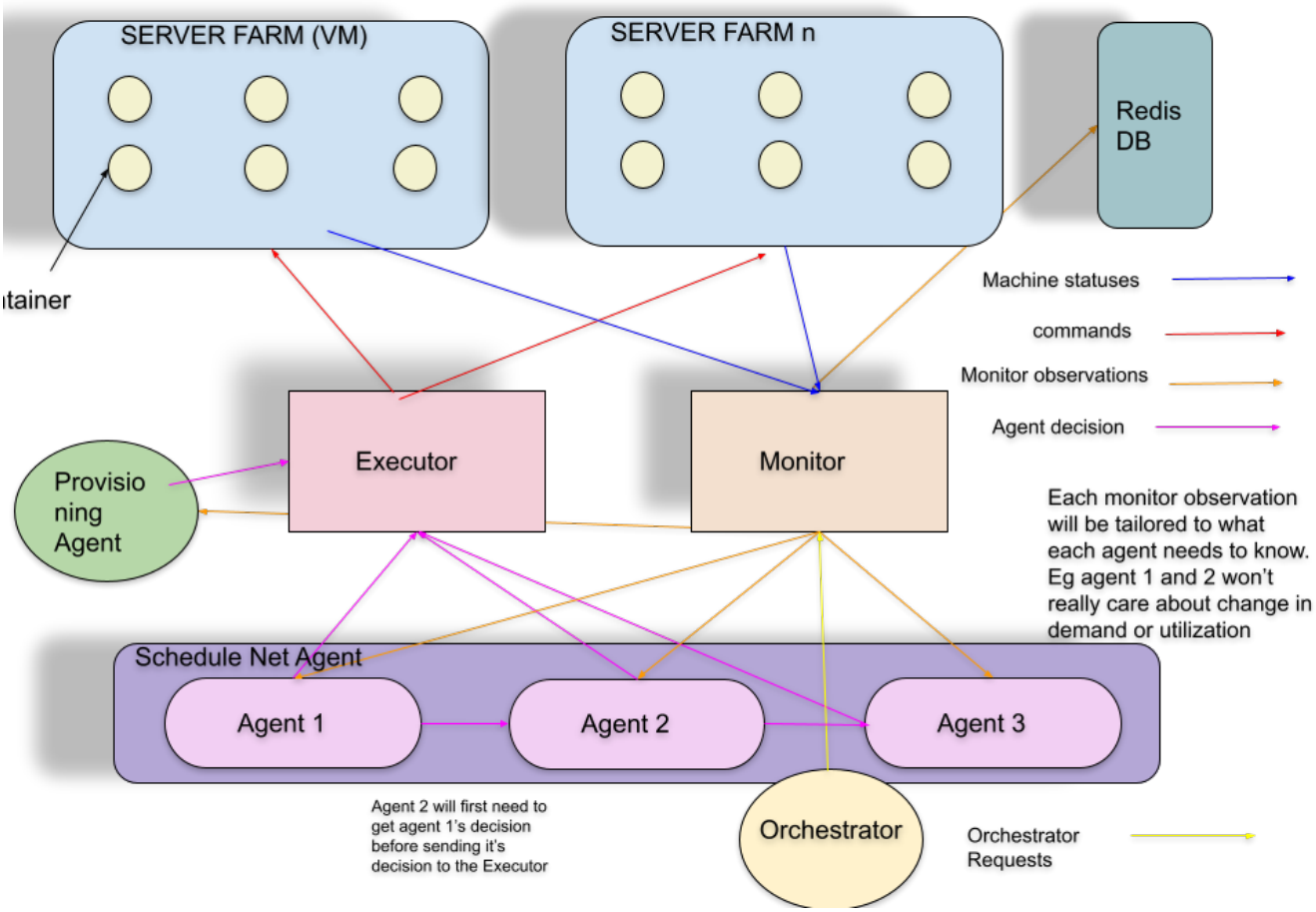

Figure 6: Bird's eye view of the project, when configured for the schedule net and resource provisioning agents

cloud service. An xlarge instance in aws costs 10 cents per hour to operate [22],a provisioning agent with a bias towards creating more instances could balloon costs. For example, if the agent only provisioned 20 instances, this would cost 2 dollars per hour or 48 dollars per day with a monthly cost up to 1,488 dollars. Even downgrading to medium would still cost 446 dollars, so the artificial cloud environment is a novel and economical way to provide the learning agents with data.

4.3.1 The Orchestrator

The orchestrator can be thought of as the boss of the entire environment, it is responsible for activating the other two processes as well as the learning agents. In addition it ensures that each process is connected with the others so that TCP communication is possible globally. Finally the orchestrator also simulates user requests by generating a custom workload class and sending the resulting requests to the learning agent(s) which then decide where those requests should go. The learning agents will send their decisions in the form of requests to the executor which will direct them to their final destination : a container running within one of the virtual machines.

4.3.2 The Observer (AKA the Monitor)

The observer has the simplest job of all the controlling process, that is to record every sent request and resource provision. Importantly it also records which ports and virtual machine/Container name is free. This task is important, especially when it comes to port numbers because if the assigned port to a container or virtual machine was always the next incremented port number, eventually the maximum possible port number would be reached. Once the max port number is reached, it would be impossible to create new containers since the subsequent port numbers would be invalid. It is also worth noting that this would all happen while there would be thousands of free ports in the system. So despite having a very simple task, the observer guarantees the efficient recycling of used ports and virtual machine/container names.

The observer communicates to the learning agents primarily via the redis DB. The agents can observer the impact they have on the environment by querying the redis DB. This is the most important task of the observer as this process yield reliable

data which allows the agents to learn.

4.3.2.1 The redis DB

When searching for a suitable database for this project, the only important factor was speed. My data would have some structure but conventional mysql databases like mariaDB or oracle wouldn't work due to them being relatively resource heavy. The solution was to use redis which supported hashes (redis equivalent of a dictionary) for structured data, but was also fast since it stores data in memory [23]. In addition to redis being fast, it is also relatively simple to use. For example when wiping the memory, the query to do that in redis is just 'FLUSHALL'. When attempting to do the same in mysql, the query is 'DROP DATABASE DB_NAME' and would have to be repeated for all databases.

4.3.3 The Executor

The executor is the most complex and busiest of all the controlling processes. The executor is responsible for sending requests where the scheduling agents want them to go. In addition it is responsible for provisioning and de-provisioning virtual machines and containers. The executor co-operates with the orchestrator during startup time and takes requests from the provisioning agent to spin up and shut down containers. The executor also keeps track of how many requests are completed/rejected as a general metric for how the system is preforming. The executor can be thought of as the learning agents' interface to the virtual machines and containers, it is through the executor that the learning agents can influence the environment.

4.3.4 The Live Environment

The live environment itself is fairly simple, its composed of virtual machines representing data-centers and containers within those virtual machines representing servers. Each container will have a simple Node.js app that will simulate actual work

being done. The app will simulate work by pausing for either 1,2,4, or 8 seconds before submitting a response. This mechanism allows for the possibility of a request timing out and will be factored in the reward function of the provisioning agent. The web app itself is made available through port forwarding. When a container is created it will be assigned a unique port, that port will then be forwarded from the same port that is on the virtual machine hosting the container. To access the app, the executor make a request to a virtual machine on a certain port and then that request is forwarded to the appropriate container. For this reason the recycling of ports was made a priority feature.

4.4 The learning agents

The learning agents can be placed in two categories : task schedulers and resource provisions. For task scheduling we have two categories of agent: zero-encoded and schedule-net. Each learning agent is based on a DQNAgent class created using pytorch. The DQNAgent is a deep learning neural net with three linear layers. The agent uses 'adam' for the optimizer and MSE (Mean Square Error) for the loss. The class can perform 3 main actions : Act,remember, and replay. When the agent Acts it takes in input data and makes a decision based on the weights of all 3 layers. Remembering is when the agent stores : the action taken at a given state, the given state, the reward resulting from that action, and the next state. Replay is when the agent applies the Q-learning algorithm, it will take a random sample from its memory buffer and adjust the layer weights based of what the samples show. For example, if the samples show that provisioning resources during periods of low utility yield a positive reward, the weights of the agent will adjust to make such decisions in the future.

4.4.1 Zero-encoded agents

A zero-encoded agent is an agent that takes in one-hot encoded data,however for resources that don't exist all dimensions are left as zero (0). One-hot encoding refers to the strategy of representing all information as a combination of ones and zeroes. For example the colors red,blue,green might be represented as : 1 for red, 2 for blue, or 3 for green. This setup saves space since the information is contained within one field, but this is difficult for Deep Q agents to learn. Instead, one-hot encoding expands this 1 field into 3 (is red?,is blue?, is green?), such that if the color is red, 'is red?' is set to 1 and the rest are 0.In other words red is equivalent to 1,0,0, while blue is 0,1,0 and green is 0,0,1. Although the zero-encoder is referred to as a singular 'agent' it is in fact two agents, one that picks the server and the other that picks the container.

The input for zero-encoded agents will typically have room enough for 100 virtual machines/containers, but only a fraction of that data will be in use. 100 was the chosen size because despite the fact that the incoming input might be of variable size, the most containers a virtual machine (of our specifications) can fit is a little less than 100. The advantage to having an agent that process data this way is that it's easy to implement. However it's not very scalable, and would not be able to be applied to any real world problem.

4.4.2 ScheduleNet

Schedulenet applies the principles from sequence to sequence learning. It organizes data into equal sized batches and processes those batches one at a time. The actual schedulenet agent itself is composed of 3 separate deep q agents. The first agent is responsible for selecting the virtual machine whose region matches that of the request. The second agent is responsible for selecting all containers with the same type as the

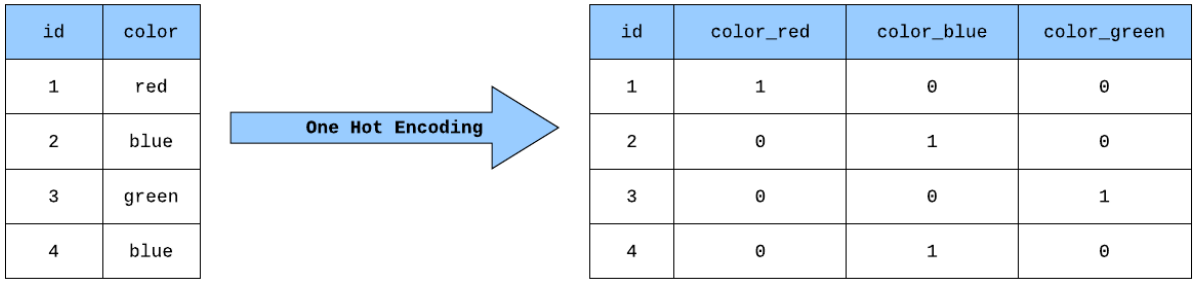

Figure 7: Example of one-hot encoding [24]

request. The third agent is responsible for selecting the destination container from the group such that the load between all containers of that type is equal. While the schedule net was more complicated to implement than the zero-encoder, it is more flexible and is thus more applicable to a real world environment.

4.5 tools

A variety of tools were used to implement this project. Several programming languages were used along with a few frameworks. Third party software was also used to support the live environment.

4.5.1 The languages

There were three language used in this project: D,TypeScript, and Python. D-lang was used for the cloud environment largely due to its speed and variety of features. Typescript was chosen for the dummy web app, partially due to a more complicated dummy app being planned. Finally Python was chosen because of the PyTorch library and its simplicity.

4.5.1.1 The D programming language

The language selected for coding the control processes of the cloud environment was Dlang. Originally elixir was considered for this task because of its fault tolerance, but for the purposes of training the learning agents, it was more important to have

an environment that delivered data fast rather than an environment that failed safely. Dlang was chosen because its speed was competitive with C and it also had many more features than the latter language. Notably, Dlang has garbage collection so there was no need to worry about memory leaks. Dlang also has a third party deep learning framework. While it was tempting to use this framework, it wasn't as mature or ubiquitous as pytorch, so if any issues arose there would not be much material online on how to fix them.

4.5.1.2 Typescript

Typescript was chosen for the node.js running in the container because initially a more complex web app was envisioned. Typescript offered strong typing and classes which would have been suitable for a more complex web application. However due to time constrains and general relevance to the project, a simpler design was chosen.

4.5.1.3 Python

Python was chosen primarily because of its rich ecosystem. There are a variety of libraries for all purposes available to download, the most important of these (for this project) are : Numpy and PyTorch. Numpy offers fast and effecient matrix manipulation which is extremely useful for deep learning. PyTorch itself is the deep learning library that offers all the building blocks and tools needed for developing a learning agent.

Another advantage of Python is its simplicity. Its syntax is easy to pick up and has been often compared to pseudo code, this makes development quick and concise. Array manipulation is another of Python's strong points. Working with array in Python is easy and Python offers an arsenal functions for editing arrays.

D was under consideration to be used for the learning agents in the beginning of the project. The reason for this was because D is faster than python, however D's

deep learning library wasn't as well supported as PyTorch was. Furthermore it was more vital that the cloud environment delivered up to date data quickly then it was for the learning agents to respond quickly to that data. In the end due to its general simplicity and rich library ecosystem, Python was chosen.

4.5.2 Frameworks

Two frameworks were used in this project: Node.js and PyTorch. Node.js was only used for running the dummy web app on the containers. PyTorch saw more extensive use with the learning agents.

4.5.2.1 Node.js

Node.js is a javascript runtime used for the backends of web applications. This framework was used to support the simple simulation web app that requests are sent to. Advantages to Node.js include the ease use as well as the large number of libraries available through npm (node package manager). The main library used was the typescript library which is just an interpreter that transforms typescript code into runnable javascript for node.js . Overall this framework was chosen due to its familiarity.

4.5.2.2 PyTorch

PyTorch is a deep learning library with a variety of building blocks and tools for developing a learning agent. One of the main reasons that PyTorch was chosen was due to its popularity. There exist many learning resources on the web with varying levels of required experience and problems arising from the framework are more likely to be answered due to the frameworks popularity. Furthermore it is exceedingly simple to mix and match parts for a deep learning agent. For example, if linear layers are not producing the desired outcome, they can be easily replaced in-line by Relu layers. Lastly PyTorch has excellent debugging tools. There are three main debugging

libraries for PyTorch : ipdb,pdb, and pycharm. The easiest of these to use is pdf which allows the user to place break points in the middle of the code and get to the root of any error.

Alternatives to PyTorch considered where : tensorflow and keras (which is a subset of tensorflow but can be used independently) for python and vectorflow for dlang. Vectorflow was discarded since there wasn't much documentation for it and if any issue arose while using it, the solution wasn't likely to be posted anywhere. In addition python is an easier base language to work with than D and in the case training deep learning agents, Dlang's speed was not as vital as it was for the cloud environment. Tensorflow is another popular AI library, but it was discarded due to its waning popularity which was being supplanted by Pytorch. Keras was another fairly simple and useful library, however since it is a subset of tensorflow, this meant it has less features than PyTorch. Ultimately PyTorch was chosen due to the latter's better debugging and ability to process larger sets of data.

4.5.3 Third party software

There were three third party softwares used in this project: Redis,virtualbox, and Docker. Redis is a NoSql database that stores data in-memory, it's advantages include that it is simple and fast. Virtualbox is the virtualization software used to spin-up and edit virtual. Lastly Docker is a containerization engine that can also spin-up and edit containers.

4.5.3.1 VirtualBox VM

A virtual machine is a separate process containing an entire computer system running within another computer. A virtual machine is created by allocating resources such as CPU,memory, and storage from the host machine and assigning those resources to a separate process, this is known as virtualization. Virtual machines allow for

multiple operating systems or apps exist on the same system. Users also have the advantage of defining the resource usage of the virtual machines. The virtual machines can be as numerous and as powerful as the host machines will allow. However virtual machines also have some disadvantages, compared to other processes virtual machines are resource heavy. In addition to edit the resources allocated to a virtual machine, the user must first shutdown the virtual machine and then edit (from GUI or command-line) which can sometimes be slow.

Virtualbox was primarily chosen due to familiarity, it does however offer a useful command-line interface. The command-line interface is very helpful because it offers a programmatic way to spin-up and deprovision virtual machines which was very useful when developing the executor process. Additionally the vritualbox command line interface can also edit network settings of a running virtual machine, which makes it easy to reach newly created containers. VMware also has a command line interface tool, however the tool was not chosen because VMware tends to be more unstable than Virtualbox.

4.5.3.2 Docker

Docker is a platform that uses OS level virtualization to create containers. While it is not the only containerization platform on the market it is by far the most popular. The popularity of the platform along with the wealth of documentation and community support made docker an easy choice for this project. One alternative considered to docker was terraform. Terraform is not a containerization platform, however it is a provisioning platform that can directly interface with all popular clout platforms. While using terraform is easy, the prohibitive costs of running many instances on AWS is why ultimately it was not chosen.

Building with docker is quick and easy, containers can be built from a massive on-

line repository containing images of various apps/run-times. If none of the repositories are sufficient for a user's needs, a docker file can be specified to pull specific images and run bash commands. Below is an example of such a file used in this project:

```
FROM ubuntu:18.04
WORKDIR /simuBack

COPY auth auth
COPY node_modules node_modules
COPY simu simu
COPY simucommon simucommon
COPY package.json .
COPY package-lock.json .

Run apt-get update
Run apt-get install -y nodejs
Run apt-get install -y  npm
RUN npm install
RUN rm simu/formats/format.txt
RUN cp simu/formats/A.txt simu/formats/format.txt

WORKDIR /simuBack/simu

RUN ls -al /simuBack/simu/formats
CMD ["node","js/start.js"]
EXPOSE 8000
```

4.5.3.3 Containers

Like virtual machines, containers are separate isolated processes that have their own allocated resources. However containers are much less resource intensive than virtual machines. Because they are lighter than virtual machines they can be spun up and shut down much more quickly. For this reason containers make a great and inexpensive alternative spinning up live instances on AWS.

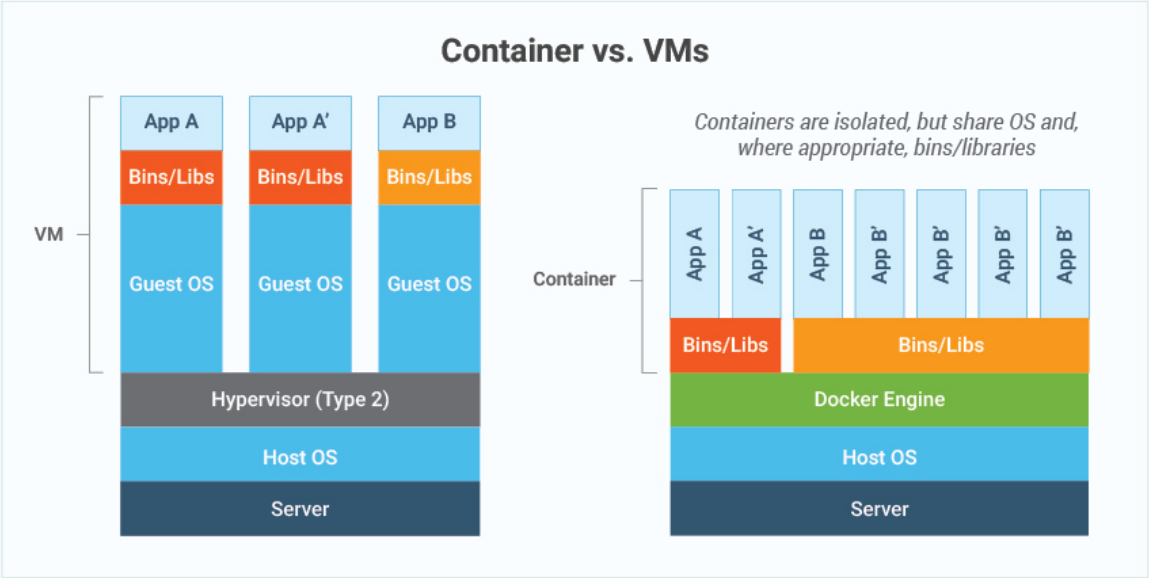

Figure 8: A container versus a virtual machine [25]

5

Challenges

5.1 Overview

Coding an artificial cloud environment and a handful of learning agents to go along with it was no easy task. It took months to develop and debug the code with the final product weighing in at several thousand lines of code. The biggest issues were typically parallelism/concurrency related. Bugs were always a scourge when encountered because they would stop the progress of development until they were resolved.

5.2 Issue: TCP Pause

Before beggining work on the orchestrator, the executor and observer were created first. During the development of the two, many test runs were executed and both processes were able to communicate with each other and run their respective commands. When work began on the orchestrator everything went well at first. The process was able to startup the executor and the observer and ensure they were fully connected,however troubles arose when trying to generate tasks to send to the executor. The orchestrator would begin generating tasks, but after a few seconds it would pause and fail to take in any additional tasks. Puzzlingly, roughly 1 out of every 15 runs would work perfectly but then proceed to fail on the next run.

This bug was by far the most difficult to solve. As mentioned above, 1 out of 15 runs would work fine but this would sometimes coincide with changes made. At first those changes seemed to be the solution to the bug but hope was quickly shattered on the second run. For two weeks straight every possible solution thrown at this bug failed, to make things worse there was no documentation for such an issue. Each of the controlling processes used a TCP server derived from github as a base. So a closer look was had at the section of code running the TCP server. It was soon found that the

reason for the pauses was due to the socket state being at -1. Negative one essentially means the socket is in a blocking state so sending any meaningful commands would send it to the queue which would get pushed down by other incoming commands and end up never running. In the end the solution was exeedingly simple, when a target socket was in the -1 state, send an empty or a blank command. This same fix was ported over to the executor.

Below is the stretch of code that ultimately solved the TCP pause issue:

```
while (true)
   {
       //auto socketSet = new SocketSet(MAX_CONNECTIONS + 1);
       //writefln("orch here0 \n");
       socketSet.add(listener);
       long sel;

       foreach (sock; reads)
           socketSet.add(sock);

       try
       {
           sel = Socket.select(socketSet, null, null);
       }
       catch (SocketException e)
       {
           socketSet.reset();
           continue;
       }
       for (size_t i = 0; i < reads.length; i++)
       {
           writefln("Socket blocking %s \n",reads[i].blocking);
           if (socketSet.isSet(reads[i]))
           {
               char[4096] buf;

               long datLength;
               //these conditional statements
               //ensure the smooth operation of the code
               if(sel != -1)
               {
```

```
     datLength = reads[i].receive(buf[]);
}
else
{
    buf = "cmd:blank,buff:buff";
    datLength = buf.length;
}
```

5.3 Issue : Parallelism bugs

Bugs relating to parallelism were the most numerous. Despite being the most numerous, they were not the easiest to solve. The bugs would at times resemble the TCP pause bug and the fixes to these bugs would sometimes cause other issues to emerge. However despite all of the bugs, D was advantageous to code in because of its 'synchronized' syntax. Any code inside a 'synchronized' block would only allow in one process at a time, this feature allowed for easy enforcement of atomicity on critical sections.

5.3.1 Race Condition

One of the main features of the cloud environment is its ability to recycle free ports. Without this feature, the environment would eventually allocate ports that don't exist because they exist the highest possible port number. To achieve this, all processes of the executor need to use a shared array of freed ports. The array's content is refreshed via a function that checks the redis database and fills the array with the database's contents. However an issue arose where a port that was already in use would appear in the freed ports array. This port would eventually be used and cause an error where the executor was not able to spin-up a new container.

There were two reasons for this error. The first reason was that the function that refreshes the free ports array was not synchronized, so more than one process could have been refreshing the array at once. A process that removed a port could

run before two processes that want to grab a new port, the other two processes can be running at the same time and pop the same port resulting in a NAT rule being written for the same port which leads to an error. The second was that the refresh function was being called too many times in appropriate places which meant that ports that were popped off the array would reappear because they had not yet been removed from that database.

The Solution to both of these issues respectively was to wrap the contents of the refresh function in a synchronized block and then remove all inappropriate usages of that function.

Below is the fixed refresh function:

```
synchronized
{
    writefln("REFRESH\n");
    string freeConQuery = format("SMEMBERS free_containers");
    Response containers = execRedis(db,freeConQuery);
    writefln("freeContainersResult %s",containers);
    foreach(k,v; containers.values)
    {
        if(!canFind(freedContainers,to!int(v.value)))
        {
            freedContainers ~= to!int(v.value);
        }
    }

    string freeServQuery = format("SMEMBERS free_servers");
    Response servers = execRedis(db,freeServQuery);
    //writefln("%s\n",servers);
    foreach(k1,v1; servers.values)
    {
        if(!canFind(freedServers,to!int(v1.value)))
        {
            freedServers ~= to!int(v1.value);
            //writefln("%s\n",v1);
        }
    }
```

```
        string freePortQuery = format("SMEMBERS free_Ports");
        Response ports = execRedis(db,freePortQuery);
        foreach(k2,v2; ports.values)
        {
            if(!canFind(freedPorts,to!int(v2.value)))
            {
                freedPorts ~= to!int(v2.value);
            }
        }
    }
```

5.4 Diagnosing Deep Learning issues.

The difficulty with diagnosing issues with the deep learning network is that problems with reward function/general learning process are practically indistinguishable from a logical error. Such was the case with the deep learning agent developed by this project. The agent itself was a combination of two other agents found via internet tutorials, the first agent had a simple design but used Keras while the other agent was more complex but used PyTorch. The idea behind the deep learning agent was to perserve the simplicity of the first agent while using PyTorch. For the most part this went well, but when training the agent, the agent would always end up choosing zero.

At first this was thought to be a problem with the reward functions, so many alterations were made but the AI would end up choosing zero. The only time the AI didn't choose zero was when a random choice was made. The reason for this error was due to a mistranslation from the keras agent to a PyTorch agent. The act function was returning the first item (0) from the forward pass, in keras this made sense since keras's forward pass returns a two dimensional array. However in PyTorch, only a one dimensional array was returned. The solution was to remove the index from the returned array.

```
    def act(self,state): #figuring out what action to take given a state
        if np.random.rand() <= self.epsilon:  #the bigger epsilon is, the more lik
```

```
        return random.randrange(self.action_size)
    act_values = self.forward(state)
    return np.argmax(act_values.detach().numpy()) #formerly return np.argmax(a
```

5.5 RAM

One other hurdle to this project was resource limitations. The computer this project was coded on had 16 gigabytes of memory. Four years ago, this was considered plenty, but in 2022 16 gigabytes is the new bare minimum. Each virtual machine spun-up consumes roughly 1 gigabyte of memory while containers consume roughly consume 10 to 20 megabytes. As the number of containers and processes increase, memory is allocated to these processes and eventually runs out causing a crash. To fix this, two more sticks of memory (8 gigabytes each) were purchased. While this action has not prevented all crashes, it has largely decreased the number.

5.6 Reward Function

The effectiveness of a learning agent hinges on the reward function. The reward function tells the agents which actions are effective in which state without having to explicitly mention each action-state combination. Calibrating and perfecting the reward function is not an exact science and sometimes simple functions tend to outperform more complex ones.

5.7 Training a Task Scheduler and a Provisioner at the same time

The task scheduler was completed before the task provisioner and performed very well when training by itself. When the provisioner was first completed it was trained simultaneously with the scheduler. The provisioner interfered with the training of the scheduler because the provisioner had a bias towards creating containers. One of the reward functions of the scheduler was partially based on the number of containers and thus began to skew towards choosing all containers as being eligible (having the same type as the task). The reason for this is because the reward for choosing an elligable

container scaled with the number of containers. It scaled so much that the reward for choosing correctly would make up for all the reward that was lost for choosing incorrectly. It was decided to train the agents seperately and tweak the scheduler reward function, however an experiment where both agents are trained simultaneously was still conducted.

6

Experiments and Results

6.1 Experiment 1: Solo Provisioner

6.1.1 Summary

In this experiment the provisioning agent is run 3 times for roughly 8 hours or 100 episodes. We measure the average reward, average container count and average container CPU/MEM util. The provisioning agent running solo means that the scheduling agent is not running with it, instead an algorithm that schedules perfectly is partnered with the provisioner. This is done as to effectively train the provisioner, as training both the untrained provisioner and scheduler at the same time does not provide satisfactory results (as the second experiment will show).

6.1.2 Results

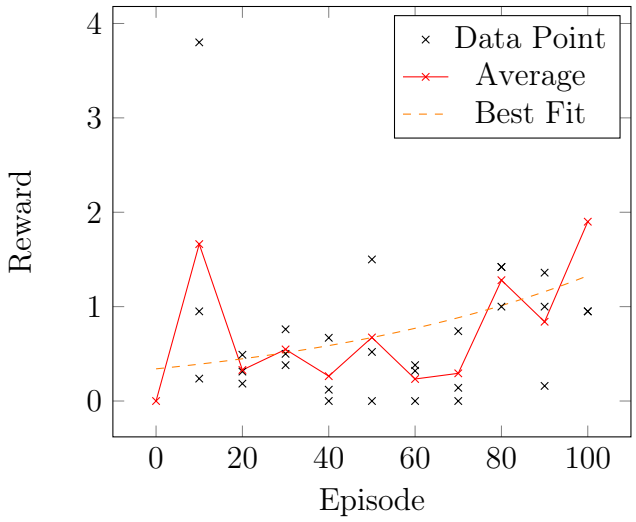

Figure 9: Reward gained by the Provisioning agent over episodes passed.

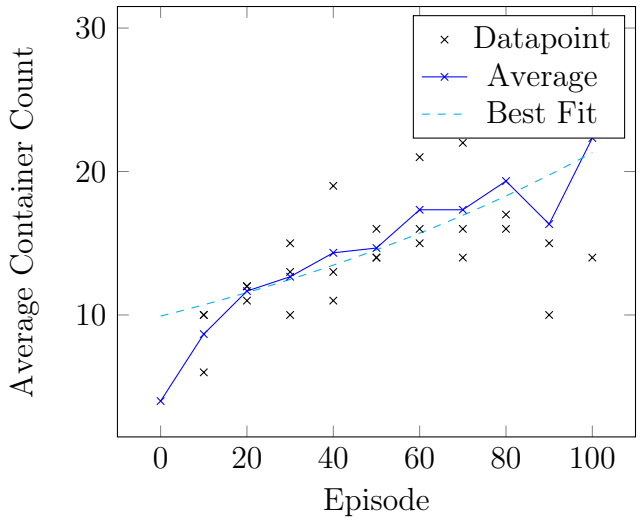

Figure 10: Average amount of containers per episode.

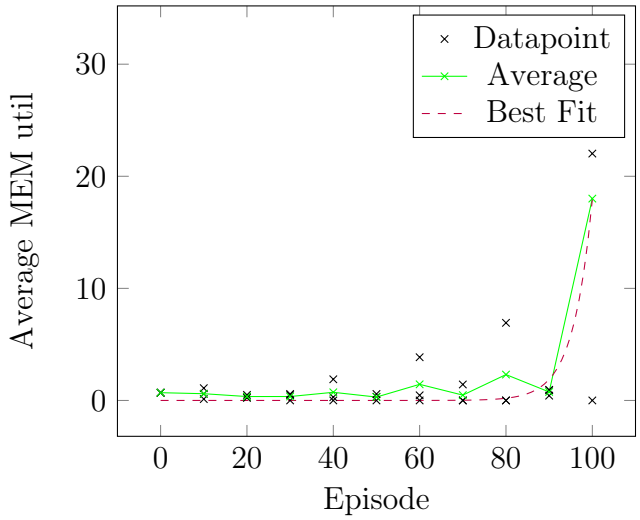

Figure 11: Average Memory util per episode.

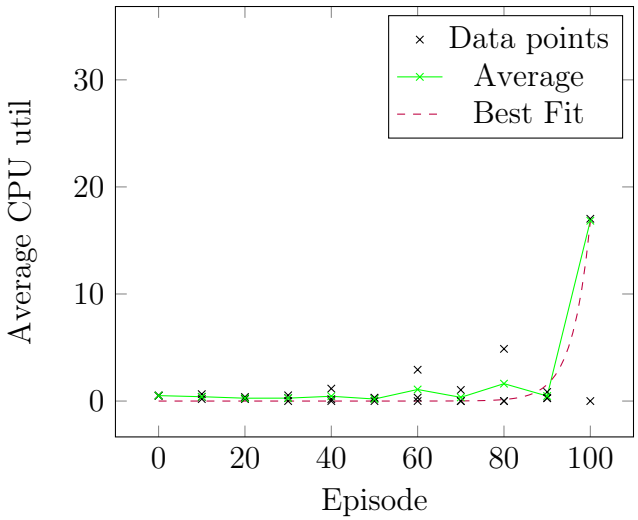

Figure 12: Average CPU util per episode.

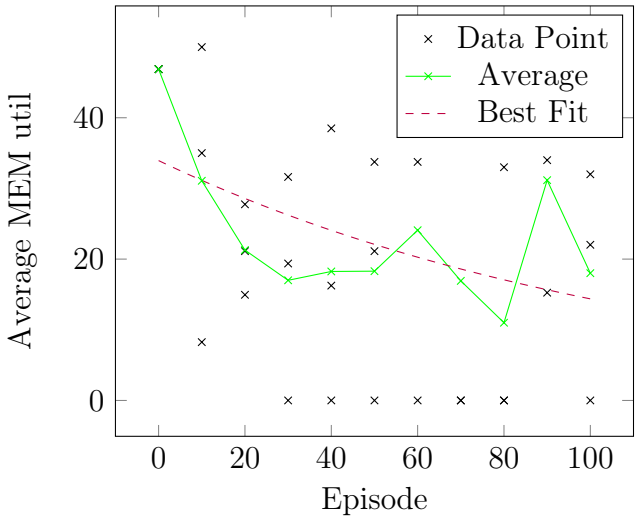

Figure 13: Average Memory util per episode corrected.

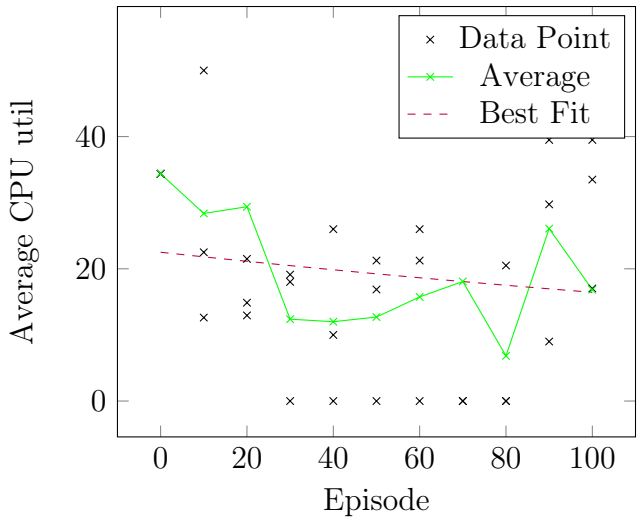

Figure 14: Average CPU util per episode corrected.

6.1.3 Interpretation

The provisioning agent was able to successfully provision resources in such a way that there were as many containers as were needed with a relatively high utilization. Figures 14 and 15 give off the sense of a low utilization, in reality when workloads are generated, they are generated for one region. This means that only one server will be working on the tasks at a time while the others are idle since a robustly trained scheduling agent or scheduling algorithm will be incentivized to send workloads to a server with the same region. When corrected for this, utilizations between 20 and 40 percent are yielded. While it may seem that corrected utilization have deteriorated from episode 0, this is because at episode 0 there will only be 4 containers in each server, naturally yielding a high utilization rate.

6.2 Experiment 2: Solo ScheduleNet

6.2.1 Summary

In this experiment the ScheduleNet agent is run solo 3 times for roughly 8 hours or 13000 episodes. This experiment yields the percentage of completed tasks per batch, reward per episode, and average consecutive reward per episode (for agents 1

and 2). Consecutive rewards for an agent refers to the cumulative award gained for consecutively correct scheduling decision.

6.2.2 Results

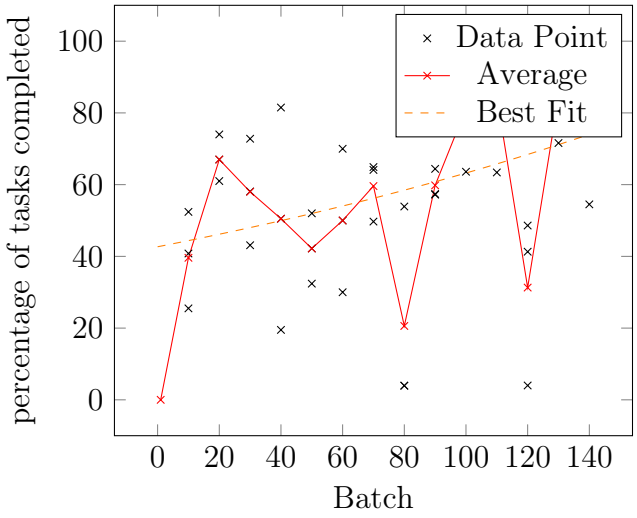

Figure 15: Percentage of tasks correctly assigned per batch.

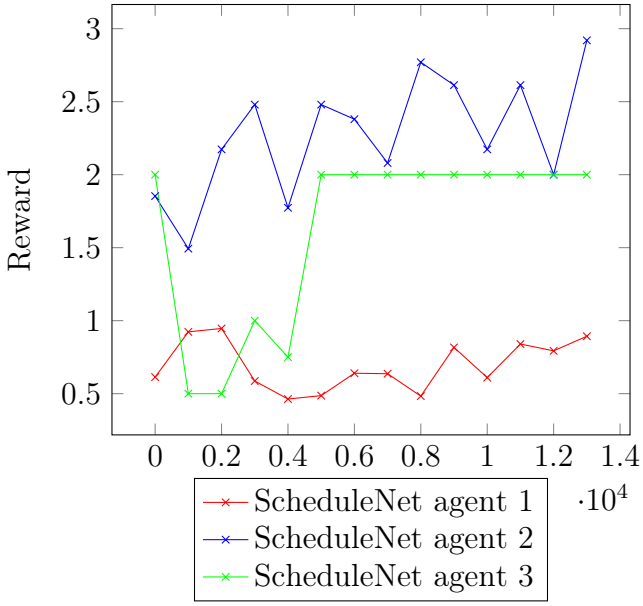

Figure 16: Reward for each agent per episode.

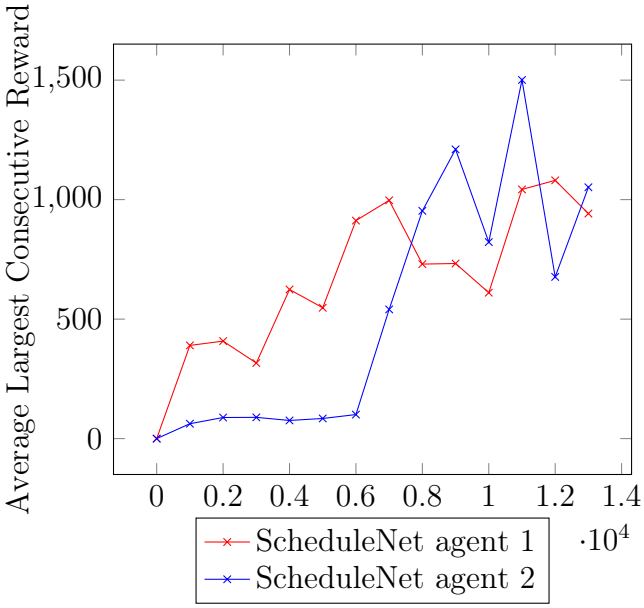

Figure 17: Average longest consecutive reward for agents 1 and 2.

6.2.3 Interpretation

The schedule net agent was successful in correctly sending tasks to where they need to go. Agent 1 which is responsible for sending the task to the correct server, became effective between the 6000th and 8000th episodes. Agent 2 which is resposible for sending the task to the correct container gradually became effective overtime and tended to reach its peak by the 11000th or 12000th episode. Agent 3 which handles load balancing between containers became effective around the 5000th episode and maintained the maximum reward thereafter.

6.3 Experiment 3: Solo ZeroEncoder

6.3.1 Summary

In this experiment the ZeroEncoder agent is run solo 3 times for roughly 8 hours or 13000 episodes This experiment yields the amount of finished tasks per batch,reward per episode, and loss per episode.

6.3.2 Results

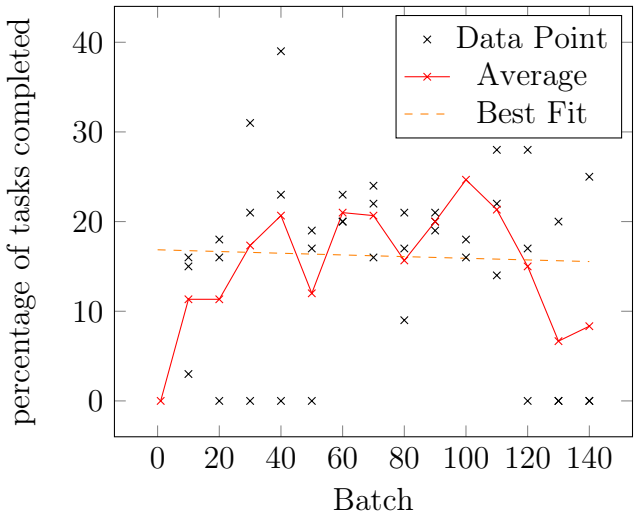

Figure 18: Percentage of tasks correctly assigned per batch.

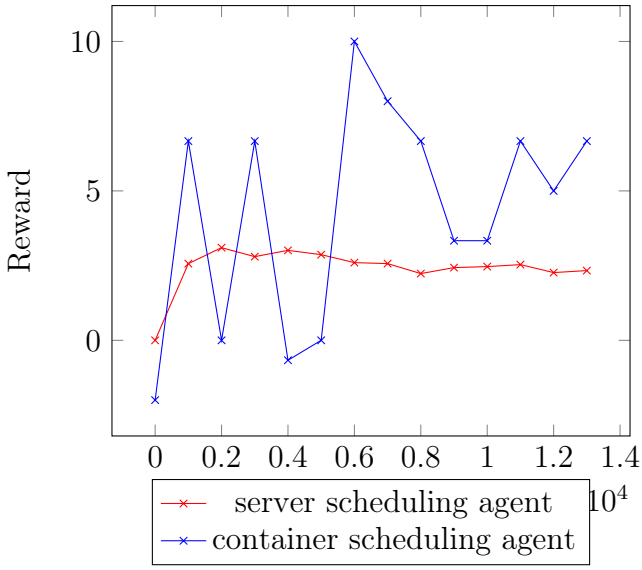

Figure 19: Average Reward per agent per episode.

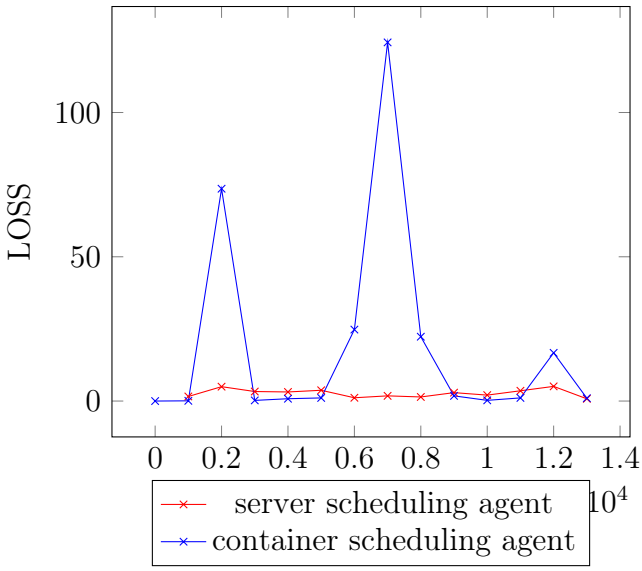

Figure 20: Loss per agent per episode.

6.3.3 Interpretation

While the zero-encoded agent saw some success in sending tasks to server, it saw less effective performance in sending tasks to containers. While the container agent seems to have performed better than agent2 in schedule net due to the reward being higher on average, its helpful to keep in mind that the reward algorithms for these two agents are different. In addition. agent 2's episodes are bigger in the sense that they have to go through all of the containers individually decide which containers not to pick. Correctly not picking a container will have a lower reward than correctly picking a container which lowers the overall average.

We can also see that the amount of tasks completed (which means they were correctly assigned) capped only at 25 percent while the cap was 80 percent for schedule net. Another issue with the zero-encoder is that when it found an action that would yield some reward, it would get stuck on choosing that option and continue to due so even when the reward is zero.

The loss for the server scheduling agent was stable and low since the server was

able to see success early on. The container agent's loss however was more unstable and began flattening near the end where it got fixated on a choice.

The the container scheduling agent of the zero-encoder fails because it tries to interpret that data from all containers all at once while the schedulenet breaks the data into digestable chunks. In addition since the servers would not be at their container capacity (given the provisioner isn't running) most of the data the container scheduling agent would read would be blank and not contain much meaning. The server scheduling agent had the same issue, but since the space for servers was significantly smaller than the space for containers in the data (since there will always be more containers than servers that host them), the server agent was able to succeed in its task.

6.4 Experiment 4: DUAL Provisioner and ScheduleNet both Untrained

6.4.1 Summary

In this experiment the provisoning agent and Schedule net are run together but are both untrained. This means that they will each be learning as the environment is run. This setup is run for roughly 5 hours or 8000 Episodes (relative to the schedulenet). In this experiment success is measured by the reward and resource utilization for the provisioner, and percentage of tasks completed and consecutive reward for the schedulenet.

6.4.2 Results

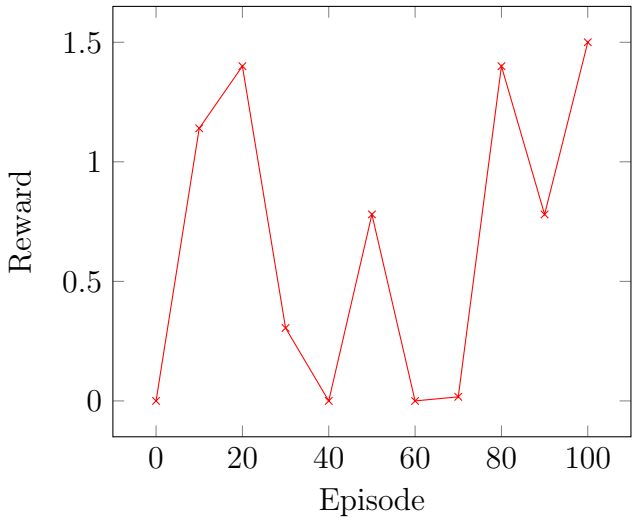

Figure 21: Average Reward for provisioner per episode.

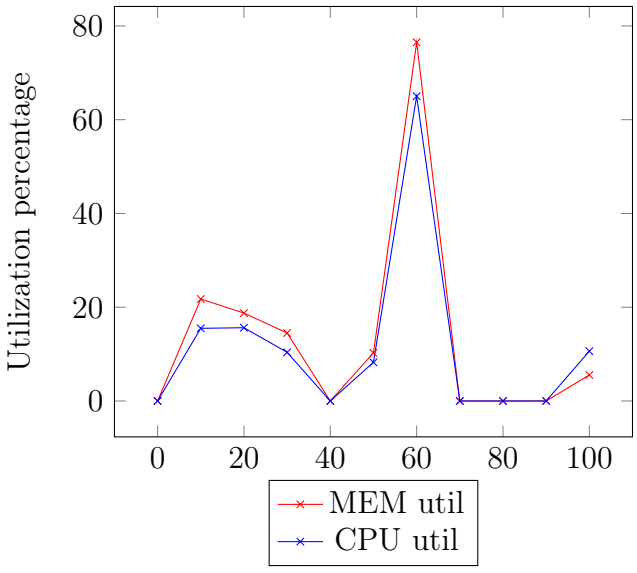

Figure 22: Average utilization per episode.

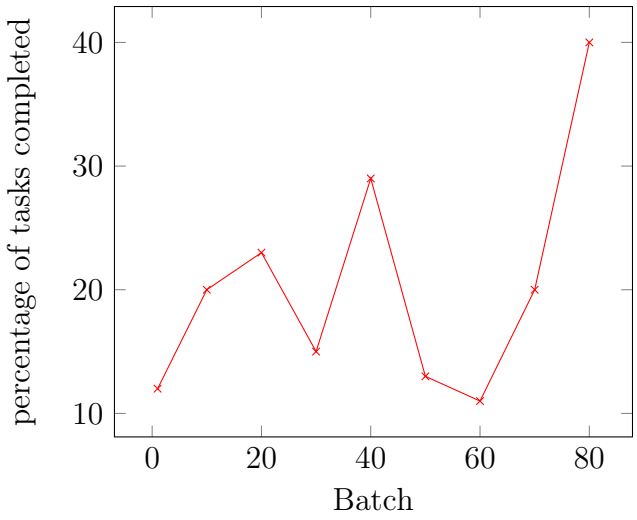

Figure 23: Percentage of tasks correctly assigned per batch.

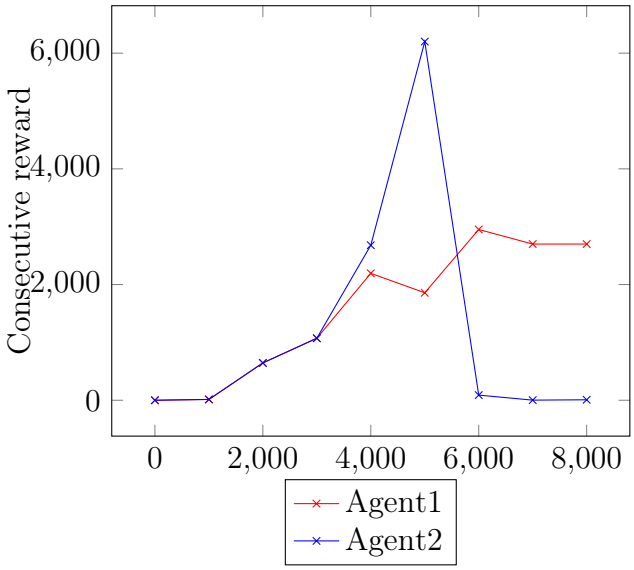

Figure 24: Consecutive reward for agents 1 and 2 of Schedulenet.

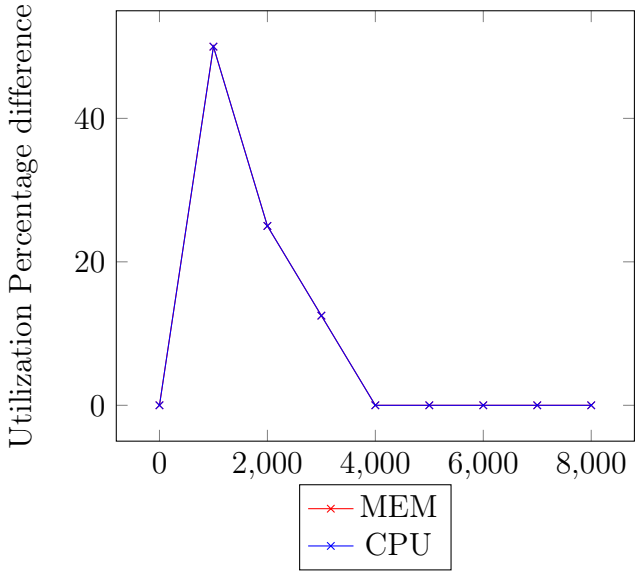

Figure 25: Utilization percentage difference for both MEM and CPU.

6.4.3 Interpretation

In this experiment we can see that the provisioner performed better than the average when compared to the provisioner-solo experiment, even reaching a peak of 80 percent utilization but falling off in later episodes while still maintaining over 10 percent. We can see however that the schedulenet agent did not perform as well. While it did see some substantial gains starting from around episode 3000, those gains quickly plummeted. The tasks completed graphs may give the impression that the agent was correctly assigning at least 40 percent of the tasks, but this is not necessarily the case. At around batch 80 a workload of many tasks with type 'A' was sent and there were many 'A' type containers within the server. At that point agent2 was not filtering any containers because under those conditions most of the containers matched the task type and automatically just choosing 1 (or approval as a elligable container) would most likely result in a reward.

Note that the consecutive reward for agent 3 was not show in figure 26 because the utilization difference between containers was relatively low through after episode

4000 of the experiment which would result in a consecutive reward that would dwarf the first two agents.

6.5 Experiment 5: DUAL Provisioner and ScheduleNet both Trained

6.5.1 Summary

This experiment is similar to experiment 4 but this time both of the agents are trained. This setup is run for roughly 5 hours or 8000 Episodes (relative to the schedulenet). In this experiment success is measured by the reward and resource utilization for the provisioner, and percentage of tasks completed and consecutive reward for the schedulenet.

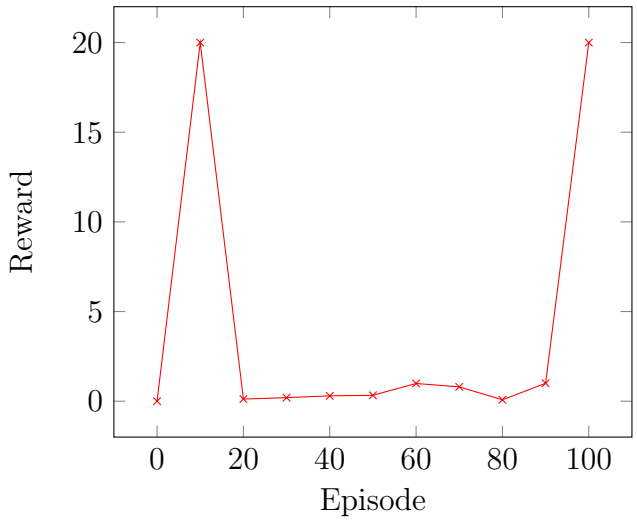

Figure 26: Average Reward for provisioner per episode.

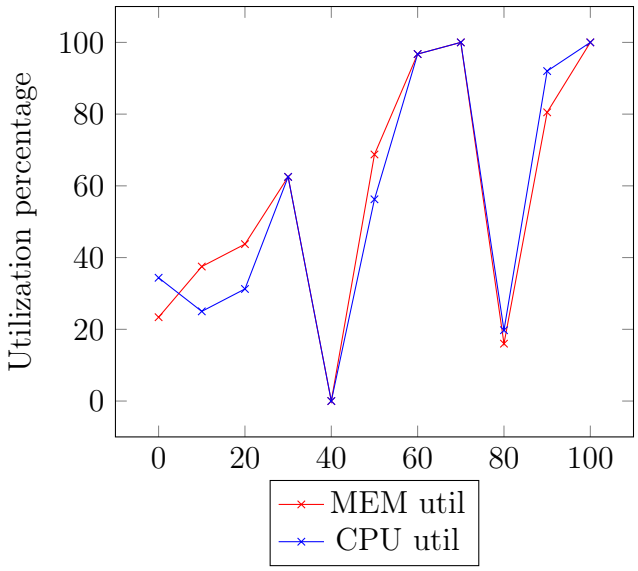

Figure 27: Average utilization per episode.

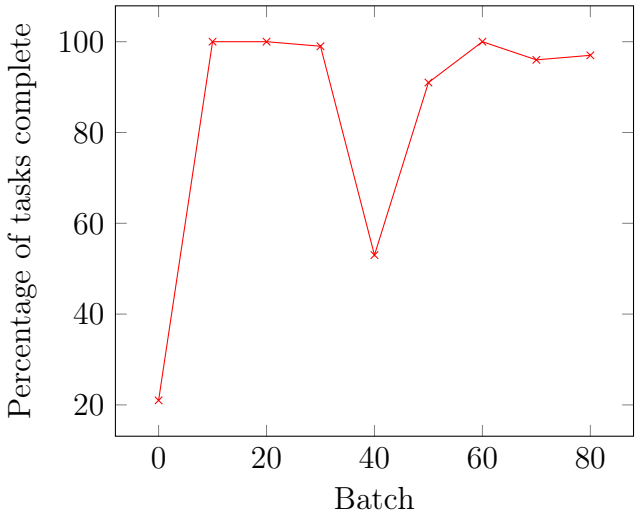

Figure 28: Percentage of tasks completed per batch.

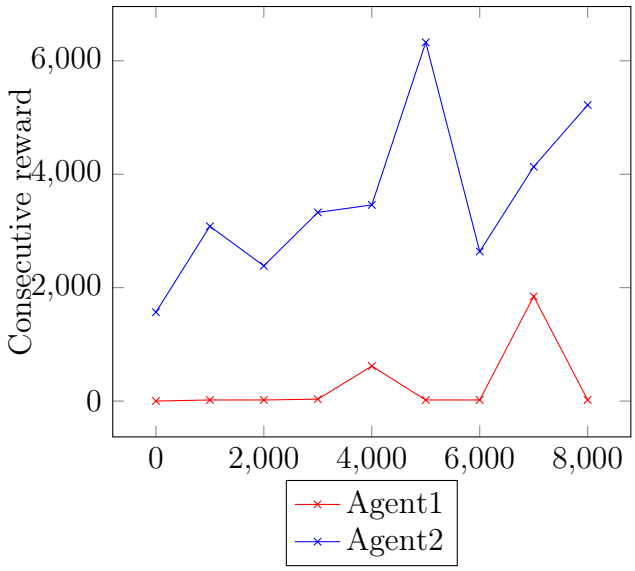

Figure 29: Consecutive reward for agents 1 and 2 of Schedulenet.

6.5.2 Interpretation

In this experiment the provisioning agent seems to have struggled while maintaining a relatively high utilization. This may be due to a reward function that is biased towards deprovisioning. However the agent's performance began picking up after the 90th episode. The schedulenet performed better than it did on the solo run, maintaining a steady 95 percent and above completion rate between batch 10 and 80 with only a drop at episode 40. The server scheduling agent looks like it performed poorly, but it usually averaged a consecutive reward of 20 with a sharp peak at around episode 7000. The container agent started out the experiment strongly and managed to improve its performance by the end, as shown in figure 31.

6.6 Experiment 6: Solo ScheduleNet with different Exploration Rates

6.6.1 Summary

In this experiment we run the schedulenet by itself 4 times for roughly 8 hours or 13000 episodes. During each run, a different exploration rate decay will be used : .995,.95,.9, and .8. Since .9 was the default for other experiments, a solo schedulenet run from the previous experiments will be imported to the results of this experiment.

The performance of the agent is measured by the percentage of tasks completed per batch and consecutive reward per episode.

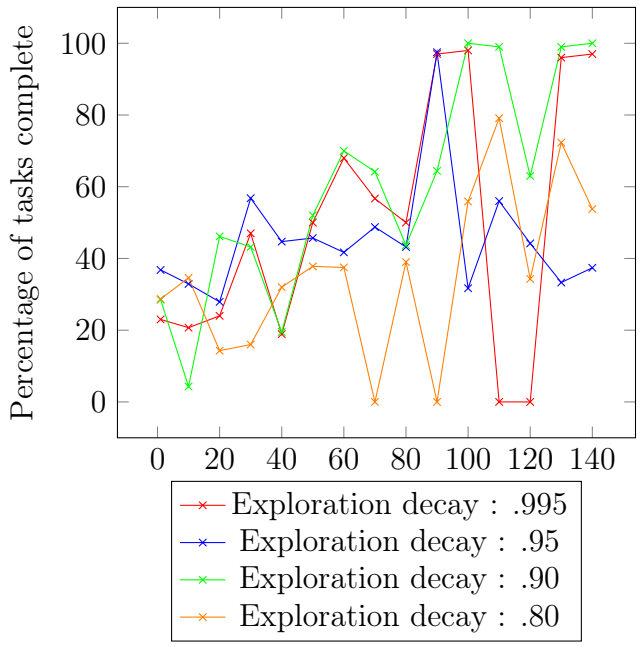

Figure 30: Percentage of tasks completed per batch.

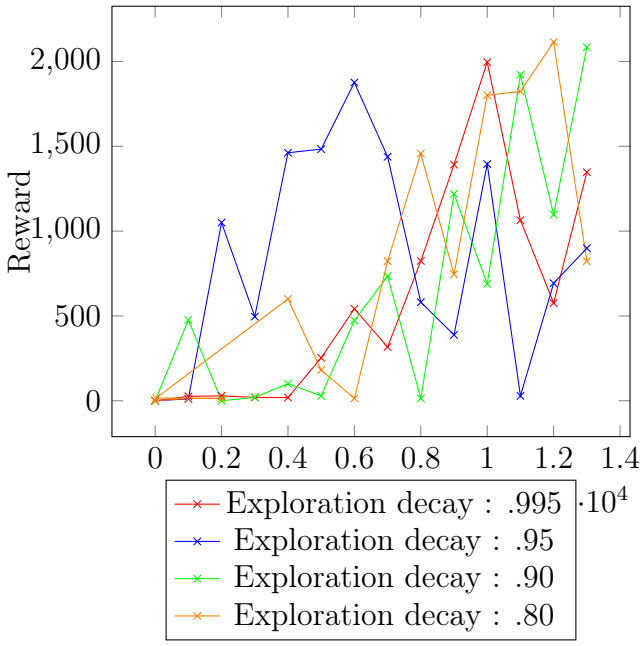

Figure 31: Consecutive reward for schedule agent1.

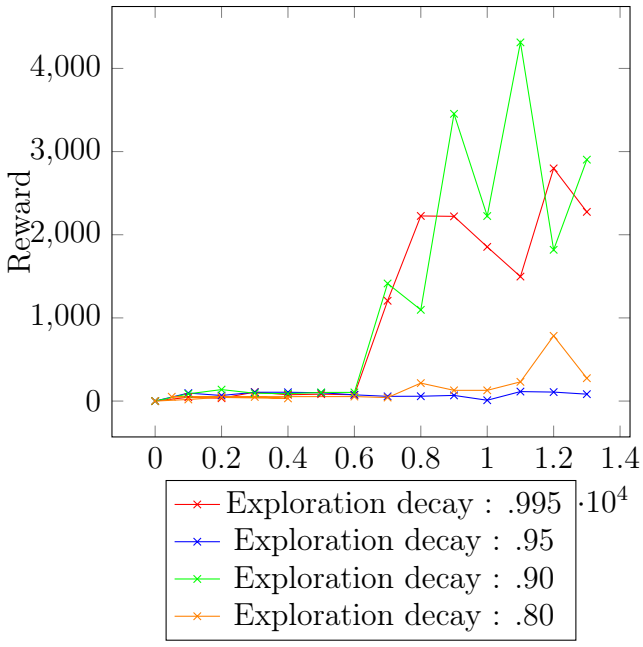

Figure 32: Consecutive reward for schedule agent2.

6.6.2 Interpretation

The results of this experiment showed that the decay rates of .995 and .90 performed the best while .95 and .80 underperformed with .80 performing the worst. Decay rates .995 and .90 inhabit a kind of Goldilocks zone with .90 performing the best. While there is not much difference between the rates of .995 and .90 for completed tasks and the consecutive reward for agent2, decay rate .90 significantly outperforms .995 for the consecutive reward for agent1.

6.7 Experiment 7: Solo provisioner with different Exploration Rates

6.7.1 Summary

In this experiment we run the provisioning agent by itself 4 times for roughly 8 hours or 100 episodes. During each run, a different exploration rate decay will be used : .995,.95,.9, and .8. Since .9 was the default for other experiments, a solo provisioner run from the previous experiments will be imported to the results of this experiment. The performance of the agent is measured by the average reward per episode and

resource utilization per episode.

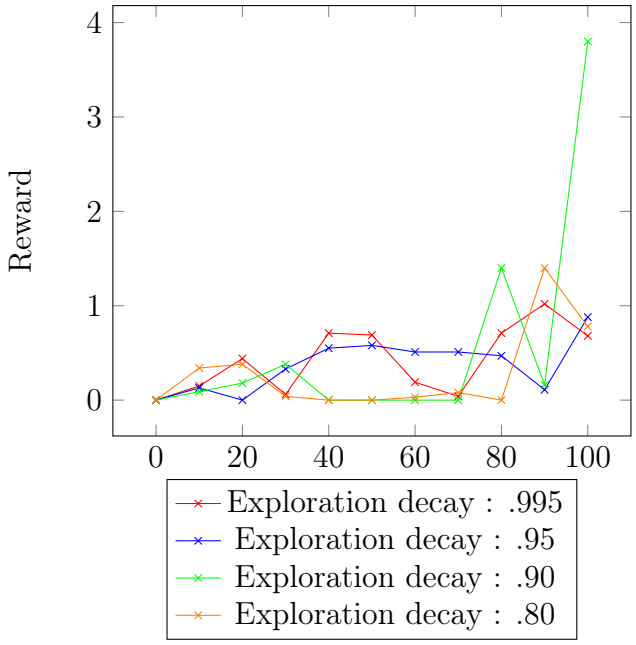

Figure 33: Average Reward for the provisioning agent for different decay rates.

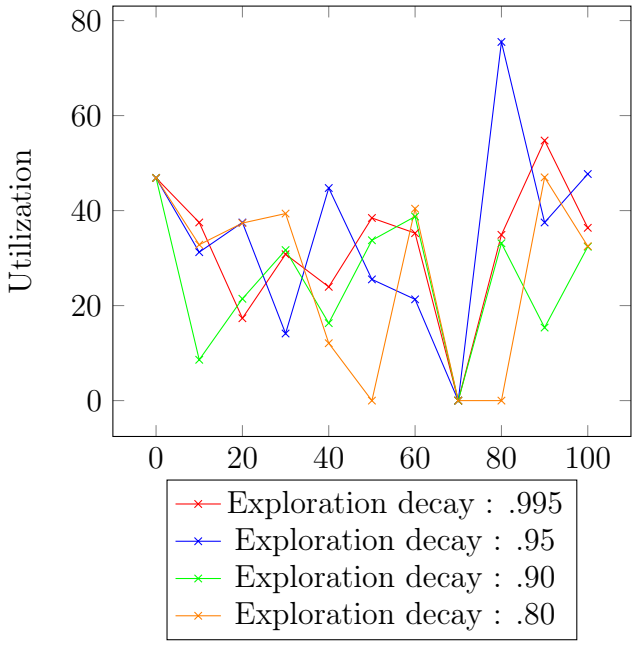

Figure 34: Corrected memory utilization for different decay rates.

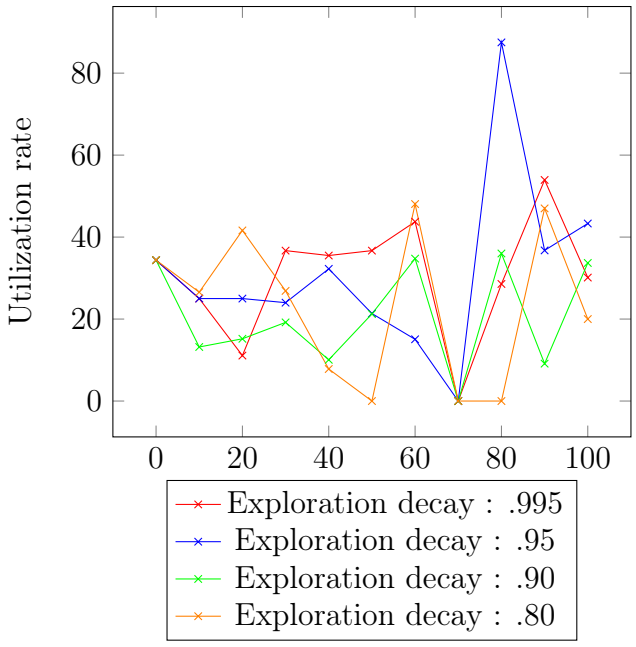

Figure 35: Corrected CPU utilization for different decay rates.

6.7.2 Interpretation

The results of this experiment were interesting. The reward graph indicates that the agent with the decay of .90 has the best performance, however, when we look at the utilization rates we see that .90 performs the worst in terms of memory usage and the second best for CPU usage. The agent with the decay rate has the best utilization rates, however its useful to keep in mind that the agent is not just scored based on utilization. The agent is primarily scored on scaling decisions made given certain utilization rates, with the actual utilization rates being used as bonus multipliers. This means in terms of scaling down with low utilization and up with high utilization, the agent with a decay of .90 made the best decisions and the lower utilization rate may be a reflection of that as they will never reach the lower or higher extremes.

7

Conclusion and Future Work

7.1 Conclusion

Deep learning agent have been used to solve problems in many different fields and industries, cloud management should be no exception. Deep learning agents when paired with reinforcement learning are a powerful tools that can respond to a given environment and make intelligent decisions. The results of this project show that deep learning agents can be used manage a cloud efficiently. Agents when trained separately and then subsequently paired together yield the best performance. While improvements can be made to the cloud managing agents, overall there is a place for deep learning in remote cloud management.

7.2 Future Work

7.2.1 Markov Game

One of issues with this project was training a provisioning and a scheduling agent at the same time. This is because the provisioning agent will change the number of containers in a server and affect the incoming reward. Markov games provide a game-theory based framework for getting adversarial or competing agents to learn and reach there respective objectives without interfering with the other. Future research and implementation of markov games would greatly benefit this project.

7.2.2 Memory

Another previously mentioned headache was memory usage. The stability of the program was affected at times by a lack of memory.A solution to this would be to purchase more memory (which was already done) or to outsource the cloud environment to actual machines that would act as the servers.

7.2.3 Expanded agent capability

One desired addition to the capabilities of the agents would be to also be able to provision servers. Each server farm/datacenter would have a fixed amount of servers that could be either be on or off, the provisioning agent would decide how many of these machines need to be on. Another capability would be for both the provisioning and scheduling agents to make their decisions based on future predicted workloads. This feature may require a third deep learning agent to make predictions and send that data to the other two agents.

LIST OF REFERENCES

[1] A. Regalado, ‘‘Who coined ’cloud computing’?’’ Oct 2011. [Online]. Available: https://www.technologyreview.com/2011/10/31/257406/who-coined-cloud-computing/

[2] M. Michael, Stelzel, J. Gardiner, C. Anderson, and G. T. Mark, *A Short History of Cloud Computing.* Springer Books, 2012.

[3] N. B. Ruparelia, *Cloud Computing.* MIT Press, 2016.

[4] P. Prajakta and B. Chiradeep, ‘‘cloud computing architecture,’’ 2022. [Online]. Available: https://www.spiceworks.com/tech/cloud/articles/what-is-cloud-computing/

[5] J. P. Mueller and L. Massaron, *Deep Learning.* O’Reilly, 2019.

[6] V. Francois-Lavet, P. Henderson, R. Islam, M. G. Bellemare, and J. Pineau, ‘‘An introduction to deep reinforcement learning,’’ 2018. [Online]. Available: https://arxiv.org/pdf/1811.12560.pdf

[7] 2022. [Online]. Available: https://pytorch.org/

[8] P. Kumar, ‘‘deep nueral network,’’ 2020. [Online]. Available: https://medium.com/@pk_500/music-genre-classification-using-feed-forward-neural-network-using-pytorch-fdb9a960a964

[9] R. S. Sutton, *Reinforcement Learning an Introduction.* The MIT Press, 1998.

[10] C. Watkins, ‘‘Q-learning,’’ 1992. [Online]. Available: https://link.springer.com/content/pdf/10.1007/BF00992698.pdf

[11] D. Karunakaran, ‘‘Q-value,’’ 2020. [Online]. Available: https://medium.com/intro-to-artificial-intelligence/q-learning-a-value-based-reinforcement-learning-algorithm-272706d835cf

[12] S. Iannuci, D. O. Barba, V. Cardellini, and I. Banicescu, ‘‘A performance evaluation of deep reinforcement learning for model-based intrusion response,’’ 2019. [Online]. Available: https://www.cse.msstate.edu/wp-content/uploads/2019/11/ic12.pdf

[13] S. Osborne, ‘‘Reinforcement learning,’’ 2019. [Online]. Available: https://www.freecodecamp.org/news/how-to-apply-reinforcement-learning-to-real-life-planning-problems-90f8fa3dc0c5/

[14] Y. Keneshloo, T. Shi, N. Ramakrishnan, and C. K. Reddy, ‘‘Deep reinforcement learning for sequence to sequence models,’’ 2020. [Online]. Available: https://ieeexplore-ieee-org.libaccess.sjlibrary.org/stamp/stamp.jsp?tp=&arnumber=8801910

[15] S. A.-A. Mahfoudh, A. B. Mohamed, A. Ahmed, and M. H. Mohammad, ‘‘A deep learning-based resource usage prediction model for resource provisioning in an autonomic cloud computing environment,’’ 2022.

[16] S.-W. Cheng, ‘‘Mape,’’ 2009. [Online]. Available: https://www.researchgate.net/figure/The-IBM-Autonomic-MAPE-Reference-Model_fig1_227108020

[17] O. Toutou, G. Hamza, and B. A. Samir, ‘‘An agent based model for resource provisioning and task scheduling using drl,’’ 2021.

[18] S. Hani, O. Hadim, B. Jaml, and M. Azzam, ‘‘Ai-based resource provisioning of ioe services in 6g: A deep reinforcement learning approach,’’ 2021.

[19] M. Huegle, G. Kalweit, B. Mirchevska, M. Werling, and J. Boedecker, ‘‘Dynamic input for deep reinforcement learning in autonomous driving,’’ 2019.

[20] A. Ashgari and M. K. Sohrabi, ‘‘Combined use of coral reefs optimization and multi-agent deep q-network for energy-aware resource provisioning in cloud data centers using dvfs technique,’’ 2021.

[21] G. Stamatakis, N. Pappas, A. Fragiadakis, and A. Traganitis, ‘‘Autonomous maintenance in iot networks via aoi-driven deep reinforcement learninge,’’ 2020.

[22] Amazon, ‘‘Amazon ec2 on-demand pricing,’’ 2022. [Online]. Available: https://aws.amazon.com/ec2/pricing/on-demand/

[23] redis.io, ‘‘Redis docs,’’ 2022. [Online]. Available: https://redis.io/docs/

[24] G. Novack, ‘‘One hot encoding,’’ 2020. [Online]. Available: https://towardsdatascience.com/building-a-one-hot-encoding-layer-with-tensorflow-f907d686bf39

[25] S. Kumar, ‘‘Container,’’ 2022. [Online]. Available: https://souravkumar.hashnode.dev/getting-started-with-docker