CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Auto-scaling cloud infrastructure with Reinforcement Learning

A comparison between multiple RL algorithms to auto-scale resources in cloud infrastructure

Master's thesis in Computer Science – algorithms, languages and logic

DANIEL EDSINGER

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2018

Master's thesis 2018

Auto-scaling cloud infrastructure with Reinforcement Learning

A comparison between multiple RL algorithms to auto-scale resources in cloud infrastructure

DANIEL EDSINGER

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2018

Auto-scaling cloud infrastructure with Reinforcement Learning
A comparison between multiple RL algorithms to auto-scale resources in cloud infrastructure
DANIEL EDSINGER

Supervisor: Peter Damaschke , Computer Science and Engineering
Advisor: Vito Cusumano, Ericsson
Examiner: Morteza Chehreghani, Computer Science and Engineering

Master's Thesis 2018
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2018

Auto-scaling cloud infrastructure with Reinforcement Learning
A comparison between multiple RL algorithms to auto-scale resources in cloud infrastructure
DANIEL EDSINGER
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

With an increasing use of cloud services for both personal and professional use, the competition for bringing the best product becomes harder as more companies provide this type of service. Not only do they want to save cost, but also improve the stability to better handle sudden, unexpected problems that can decrease the performance and responsiveness of their cloud service. Therefore, the purpose of this project was to propose and evaluate different solutions that can auto-scale the cloud infrastructure based on its resource usage. Also included in the report are algorithms that did not provide any usable results or could not handle the complexity of the problem. We developed three different reinforcement learning algorithms in Python, using the Tensorflow framework to train neural networks, and compared their performances in terms of both cost and stability. These algorithms were implemented to work on virtual machines with Apcera installed and were trained with data collected through Apceras API. The training was done in a simulation of the cloud cluster. The results of this project shows a noticeable difference between these three algorithms. While all three work to some degree, one stands out and performs significantly better than the other two in terms of cost and the stability of the cluster. Conclusively, we have an algorithm that can accurately predict how to scale the cloud cluster based on the time of day, and the current resource usage.

Keywords: Computer, science, Q-learning, SARSA, machine learning, reinforcement learning, cloud computing, EC2, AWS, Apcera.

Acknowledgements

I would like to thank Vito Cosumano and Per-Johan Wiberg for giving me the opportunity to work at Ericsson. The help and feedback from them and the others in their team helped me structure my master's thesis into something I thought was fun and interesting.
I also want to thank my supervisor, Peter Damaschke, who throughout the project helped me formulate and rethink my writing. His input has been really helpful for writing this paper.

Daniel Edsinger, Gothenburg, December 2018

Contents

List of Figures

List of Tables

1

Introduction

This chapter will go through the motivation behind this project, its goals and limitations.

1.1 Background

Cloud computing services such as Amazon Web Services (AWS) are easily accessible and can be used for computational power, database storage, applications and other IT resources through a cloud services platform via the internet with pay-as-you-go pricing. These services allow one to rent one or multiple virtual machines (VM) to run their own programs or services. In order to keep performance high and avoid slowdowns, it is important to regulate the computational power between these VMs, e.g. keep all resources such as CPU, memory, disk, and network from maxing out.

The objective of this project is to work with Ericsson's cloud service to investigate, evaluate and identify opportunities and challenges for applying machine learning, artificial intelligence, or similar technologies in their cloud platform environment. The environment consists of a cluster of Instance Managers (IM) which are part of a run-time environment for Docker containers, a packaging software for deploying and running jobs/apps [1]. Each IM is a VM rented from AWS with the Apcera platform installed and is where these Docker containers are deployed. The use-cases that this thesis will focus on are the following:

- Distributing the workload on containers across the system
- Auto-scaling of applications and infrastructure

The current method of assigning new instances of a container to an IM with Apcera is based on a score calculated from available CPU, disk, memory, and network. All IMs then immediately return that score and the best IM is selected to run said instance. It will not take into account any fluctuations in CPU, disk or network usage of said jobs over time, nor can a job transfer between IMs to better distribute the load. This problem is connected to the first use-case.

For the moment, Ericsson's cloud service only has one type of VM that they rent from AWS while there are many more configurations available [2]. There could instead be an option to add or remove an IM with different resource configurations and cost. Exactly what type to remove/add depends on how much resources are

used across the cluster at any given time. This problem is connected to the second use-case.

In summary, these two use-cases can help the cluster handle under-/over-utilization of one or multiple IMs. The solutions to these problems are meant to address issues on the servers such as responsiveness and resilience of the system.

The solutions are meant to work with Ericsson's connected vehicle ecosystem and are expected to improve the overall robustness of the system, reduce down-time and increase the level of automation. Ideally, the final algorithm will result in a system that is fully automated and automatically scales resources up/down based on demand. For this thesis, it is important to explore the different technologies in relation to each use-case and to list the pros and cons of each method.

1.2 Context

In a survey made by Chandola, Banerjee, and Kumar, they investigated how different techniques in machine learning could be used while searching for different types of anomalies [3], such as Intrusion Detection, Industrial Damage Detection, etc. The investigated techniques have multiple designs with both advantages and disadvantages. This survey does not investigate exactly the kind of data that we will use, but it will help give an understanding on what options are available. It is possible that we can correlate a detection technique they used for a specific area, to one that we will use in this project and directly translate that.

Teodoro, Verdejo, Fernández and Vázquez studied anomaly detection focusing solely on network intrusion detection [4]. Here they discuss what techniques are appropriate for network intrusion. Just like in their study, we would have to adjust what kind of algorithm we use to solve a specific problem.

Reinforcement learning (RL) is a known technique when optimizing the workload of VMs [5] [6]. While these two papers in the previous paragraphs focus explicitly on read/write (I/O) and network performance for VMs, this project will include the optimization of CPU, memory and disk resources over a whole cluster. How these clusters work and look is described in more detail in Section 3.4.

1.3 Goal

The goal of the project is to investigate different types of algorithms and evaluate if they are suitable for the concerned use-case described in Section 1.1. We will list the pros and cons for each approach and propose an appropriate solution to our problem. This includes an analysis of availability of input data, expected accuracy, efforts needed to implement it and how resource (CPU/Memory) demanding it would be. This also includes investigating whether or not machine learning is in fact the right way to go for all use-cases, or if some of them can be solved using simpler statistical models.

The same use-case could be implemented in many different ways (e.g. simple logic in bash scripts, statistical models, machine learning, deep neural networks etc.). The question is to figure out which is the appropriate way.

However, because the data collected from the cluster lack any sort of annotations, there are limitations on what methods can be used. Methods using supervised learning are not capable to solve this problem, as they are designed to classify each input [7], but also need annotations to learn. Therefore, RL is more suitable as it can learn using rewards and punishments without knowing exactly *how* a problem should be solved [3].

The different methods tested/evaluated in this project are several RL algorithms: Q-Learning [8], SARSA [9], Double Deep Q-Learning [10] and AC3 [11]. The comparison between these algorithms will include their speed, cost and if they made any improvements on the system. They will all have the same state inputs, reward function and possible actions to take. Their differences will be how they predict *good* actions at each state.

The information required to test the algorithms consists of different configurations of how many IMs and job instances are currently running, and how much resources are used at any given time. This data can be collected from the real configuration in Ericsson's system.

1.4 Limitations

This project will only test the solution on a simulation instead of a real system. The simulation environment will have the additional functions of choosing where to put a container and transferring them between VMs. While Apcera has no such feature, those functionalities can be found in a similar program called Kubernetes when using *nodeselector* [12].

It is important to understand that with reinforcement learning, as the environment state-space increases, the complexity of the problem will also increase. In this project, state-space will depend on the number of VMs and containers. That is why Deep Q-learning is a suggested algorithm in Section 1.2, as the problem can become too complex for the Q-learning and SARSA algorithms. On the other side, it is harder to change the input and output for the neural network in Deep Q-Learning, e.g. if the number of VM types change, the network will have to re-train everything.

To clarify the scope of this project, the following problems will not be explored in this project, but are still related to the subject:

- Preemptive fault detection
- Automatic fault recovery
- Automatic resource profiling of applications
- Other anomalies or deviations on the platform

2

Theory

This chapter will describe the theory requisites needed to understand the different methods and technologies that are used in this project. Specifically how we can train a computer to make decision in order to achieve the best outcome. It will also explain a common technique applied to problems that otherwise would be to hard or slow so solve because of the its complexity.

2.1 Reinforcement learning

As stated in Section 1.3, RL can solve problems using a function as a reward or penalty. An agent interacts with an environment by carrying out actions, then receives rewards depending on how "good" the actions were. How this works can be seen in Figure 2.1.

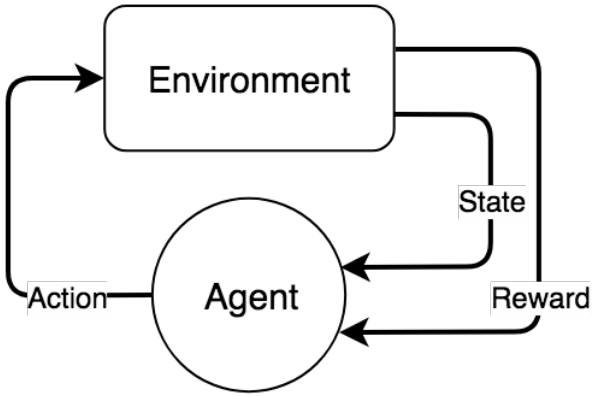


Figure 2.1: How an agent uses actions on an environment.

Each action the agent takes will result in a new state and a reward depending on how *good* that state is. These rewards will reinforce good behaviors and the agent learns what actions will result in the highest reward.

2.1.1 Markov decision process

In RL, problems are modeled as a *Markov decision process* (MDP) [13] [14]. The environment can then be modeled as a transition probability between states s and

s' with $P_a(s, s')$. The probability depends on the current state s and each action a that can be taken at that state. The MDP model of an environment consists of a tuple $\{S, A, P_a(s, s'), R_a(s, s'), \gamma\}$ whose components denote the following:

- S a finite set of states
- A a finite set of actions
- $P_a(s, s') : S \times A \times S \to [0, 1]$ the probability of reaching state s' with action a
- $R_a(s, s') : S \times A \times S \to \mathbb{R}$ the immediate reward of taking action a in state s, reaching state s'
- $\gamma \in [0, 1]$ the discount factor of future rewards.

The reward function $R_a(s, s')$ depends on both the agents current state before the transition, and the action it chooses. Performing a specific action a at state s will result in the agent reaching an arbitrary state s' with a reward. The MDP can be defined with rewards only at terminal states, on all states or no rewards at all. Although, an MDP model with no rewards is already solved as there are no paths that can result in a bigger rewards, hence it is already optimal.

For an MDP model, there is an agent that will act upon it, taking actions that cause transitions between states. Figure 2.2 shows how the agent uses actions to get rewards. At each state s, when an action a is taken by the agent, the environment will return a new state s' and a reward $R_a(s, s')$.

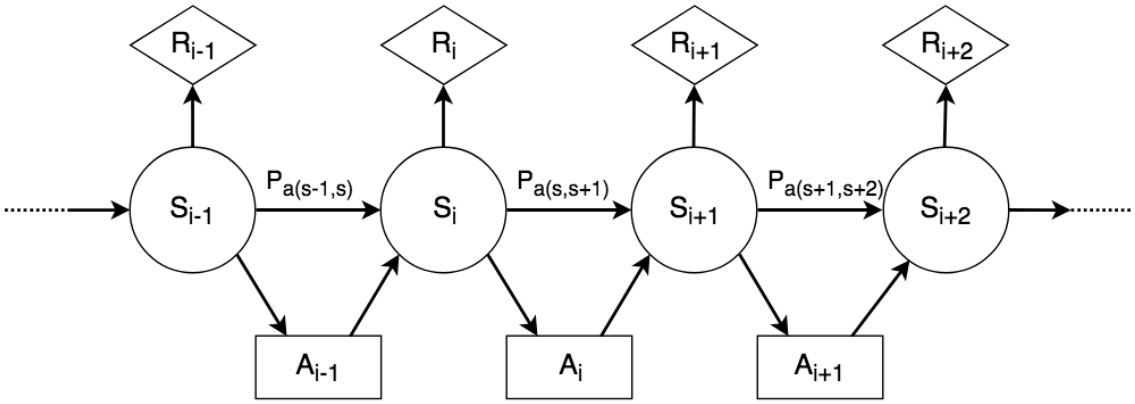


Figure 2.2: An MDP with transitions from state to state, each with a reward R. P is the transition probability from state s to s' with action a.

2.1.2 Policy

A policy π describes an appropriate decision for any state s in the form of state-action pairs. The goal of using an MDP is to find an optimal policy π^* which predicts a "best" path with the highest possible expected reward [15]. That is, from an initial state s, it will achieve the highest reward possible. The total reward is the sum of all rewards $R_a(s, s')$ from all states that are visited.

We define π as the general solution to the MDP model, and $\pi(s)$ as solution to any specific state s. The policy function $\pi(s)$ will generate an action made by its policy

π so that the decided action a is the value generated from the function value $\pi(s)$, for both the reward and probability function $R_a(s, s')$ and $P_a(s, s')$. These functions can therefore instead be written as $R_{\pi(s)}(s, s')$ and $P_{\pi(s)}(s, s')$ for any state s using an arbitrary policy π.

For each $s \in S$ there exists a function value $V(s)$ which is defined as the expected future reward for state s. The expected value function following a policy π can be written as an infinite-horizon discounted model [14]:

$$V^{\pi}(s_t) := \Big[R_{\pi(s_t)}(s_t, s_{t+1}) + \gamma R_{\pi(s_{t+1})}(s_{t+1}, s_{t+2}) + \cdots + \gamma^n R_{\pi(s_{t+n})}(s_{t+n}, s_{t+n+1}) + \ldots\Big] \tag{2.1}$$

We can observe in Equation 2.1 that since the the discount factor γ^n is exponential, the expected future reward becomes smaller as the number of steps increases, unless γ is 1. The same function can also be written as:

$$V^{\pi}(s_t) := \left(\sum_{t=0}^{\infty} \gamma^t R_{\pi(s_t)}(s_t, s_{t+1})\right) \tag{2.2}$$

Although this function describes the solution to the MDP, we need a way to solve the problem in finite time. To describe the function as finite, we rewrite $V^{\pi}(s)$ and introduce the probability function $P_a(s, s')$:

$$V^{\pi}(s) := \sum_{s'} P_{\pi(s)}(s, s') \times (R_{\pi(s)}(s, s') + \gamma V(s')) \tag{2.3}$$

Here $R_{\pi(s)}(s, s')$ is the immediate reward for performing an action a at state s determined by a policy π, and $\gamma V(s')$ is the discounted expected future reward for reaching the next state. Note that $\sum_{s'} R_{\pi(s)}(s, s') + \gamma V(s')$ equals the infinite-horizon model in Equation 2.1 and 2.2 but is solvable in finite time, as $V(s')$ is the reward for all future states and $R_{\pi(s)}(s, s')$ for the current state.

The variable γ decides how much influence future states have on the current state value. A lower γ means a lower reward from the future states.

We also have to define the policy function in order to complete the value function:

$$\pi(s) := \operatorname{argmax}_a \left\{\sum_{s'} P_a(s, s') \times (R_a(s, s') + \gamma V(s'))\right\} \tag{2.4}$$

To compute the value of a state s ($V(s)$), we need to know the already calculated value of $V(s')$ for the next state, so it is necessary to store all state-values in an array the size of the state space S.

In order to find the optimal policy π^*, we need to iterate the value function $V(s)$ for all states. That is possible by using the previous value function V^{i-1} to update the next iteration of the function V^i. The value iteration loop can be seen in Algorithm 1. This will simulate the infinite horizon discount reward in a finite amount of time.

Algorithm 1 Value iteration

1: $V^0 :=$ arbitrary value function for all states
2: $i := 0$
3: **repeat**
4: $\quad i = i + 1$
5: $\quad$ **for** each state $s \in S$ **do**
6: $\quad\quad V^i(s) = \max_a \left\{ \sum_{s'} P_a(s, s') \times (R_a(s, s') + \gamma V^{i-1}(s')) \right\}$
7: $\quad$ **end for**
8: **until** $V^i - V^{i-1} < threshold$

The value iteration will continue until the difference between V^{i-1} and V^i has reached a certain threshold. The threshold is generally set to a small number but can be altered depending on the problem. Given enough time and value iterations, the policy will converge as long as no state is excluded from the value and policy iterations.

The optimal policy theorem was proved by Ross in 1983 [15] [16], and there exist at least one optimal policy solution π^* so that we also have an optimal value function $V^*(s) \equiv V^{\pi^*}(s)$ [8]:

$$V^{\pi^*}(s) = \max_a \left\{ \sum_{s'} P_a(s, s') \times (R_a(s, s') + \gamma V^{\pi^*}(s')) \right\} \tag{2.5}$$

Note that this method only is possible if the whole model is known. We cannot perform value iteration if the reward or probability function are not given. These become harder to provide as the complexity of the problem increases.

2.1.3 Q-learning

We can solve small MDPs fast and reliably when the entire MDP model is available. It can be solved by finding a greedy policy π for the problems.

The name "greedy" comes from the policy always choosing the action resulting in the highest reward in the following state, as seen in Equation 2.4. As such, we update the value function with "greedy" actions when performing value iterations. When the model is known, we can update all the expected rewards for all states, also called value iteration, and ensure that the policy is optimal. This method is however not very practical since for most problems we are trying to solve, the model is not known.

2.1.3.1 Q-values

Q-learning is a form of model-free RL [8] and introduces an ϵ-greedy policy written as π', and state-action pairs $Q(s, a)$, also called q-values, for choosing actions. The q-values $Q(s, a)$ for a state s is a collection of the expected reward $V^{\pi'}(s)$ for performing different actions. E.g., if there are four possible actions at state s_t, then $Q(s_t, a)$

consists of four different values. The required size of the array to store all values, is the state space S times the action space A. Equation 2.6 shows how these q-values are calculated.

$$Q^{\pi}(s,a) = R_a(s,s') + \gamma \sum_x P_a(s,x)V^{\pi}(x) \tag{2.6}$$

where for the immediate reward $R_a(s,s')$, the state s' is the outcome of taking action a at state s, likewise for x.

We can say that the highest expected reward from a state s is the q-value of $Q(s,a)$ with the argument a that yields the maximum value:

$$V^*(s) \equiv V^{\pi^*}(s) = \max_a \left\{ R_a(s,s') + \gamma \sum_x P_a(s,x)V^{\pi^*}(x) \right\} = \max_a Q(s,a) \tag{2.7}$$

If the outcome of performing an action a at state s is always the same, i.e. the probability function $P_a(s,s')$ returns the same value for any states and actions, then we can simplify and rewrite Equation 2.6 by replacing the value function:

$$Q^{\pi}(s,a) = R_a(s,s') + \gamma \max_a Q(s',a) \tag{2.8}$$

2.1.3.2 ϵ-greedy policy

The procedure of the ϵ-greedy policy is the following:

- With probability $\epsilon \in [0,1]$ take a uniformly random action.
- Otherwise choose the best action according to the estimated state-action pairs from the q-values $Q(s,a)$.

Just like the normal policy π from Section 2.1.2, π' chooses an action based on the value function $V^{\pi'}$. The modification is to take a random action with a probability of ϵ. The ϵ variable helps the algorithm explore paths that the policy otherwise would not choose, and it often decreases over time as the training progresses.

In a model-free MDP we only know how many states there are, not the reward or probability function $R_a(s,s')$ and $P_a(s,s')$. Therefore we can not perform value iteration, but instead have to go from state to state in order to update the expected reward in forms of episodes.

These episodes are training periods made to update the value function by trial and error. Each episode will start at state s_0 and traverse between states with actions chosen by the policy π', while updating the q-value function for each state it visits. When finally reaching a terminal state, it starts over and resets the model while keeping the updated q-value function. The cycle continues for M number of episodes and will perform the following steps:

- A state s_n in the N:th episode.
- Choose an ϵ-greedy action a_n to maximize the reward.

- Update $Q_n(s_n, a_n)$ with the expected reward from the earlier episode $n-1$ with a learning rate α. This is shown in Equation 2.9.

$$Q_n(s_n, a_n) = \begin{cases} (1-\alpha)\underbrace{Q_{n-1}(s_n, a_n)}_{\text{old value}} + \alpha \overbrace{(\underbrace{R_a(s_n, s'_n)}_{\text{reward}} + \gamma \underbrace{\max_a Q_{n-1}(s'_n, a)}_{\text{future reward}})}^{\text{learned value}} & \text{if } s \text{ is not terminal} \\ Q_{n-1}(s_n, a_n) & \text{else} \end{cases} \tag{2.9}$$

2.2 Artificial Neural Networks

An Artificial Neural Network (ANN) consists of a network of neurons connected to each other. These neurons are placed in three different layers: input layer, hidden layer, and output layer, as seen in Figure 2.3. The hidden layer can consist of multiple neuron layers with different types of connections. For a fully connected network, every neuron in a layer passes its value to each neuron in the next layer.

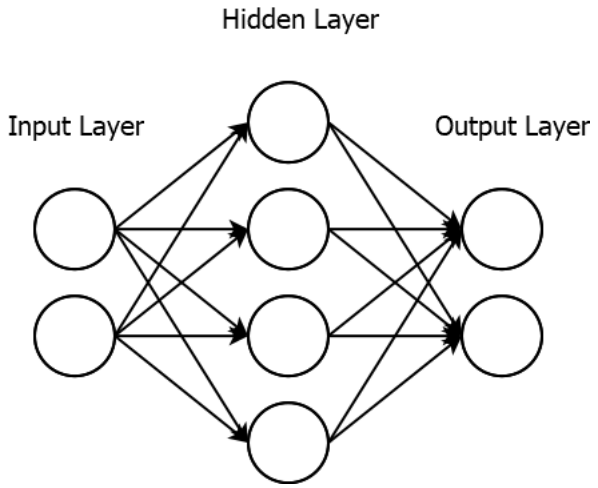


Figure 2.3: An Artificial Neural network. The hidden layer consists of several layers of neurons.

Figure 2.4 shows a closer look inside the neural network (NN). The value y inside the last node is the sum of all inputs times their respective weight w so that $y = x^\intercal W$, where W is the collection of all weight, but with the addition of an activation function ϕ we get the output value $\hat{y}$ as seen in Equation 2.10. The output value is then sent to all other connected neurons in the next layer.

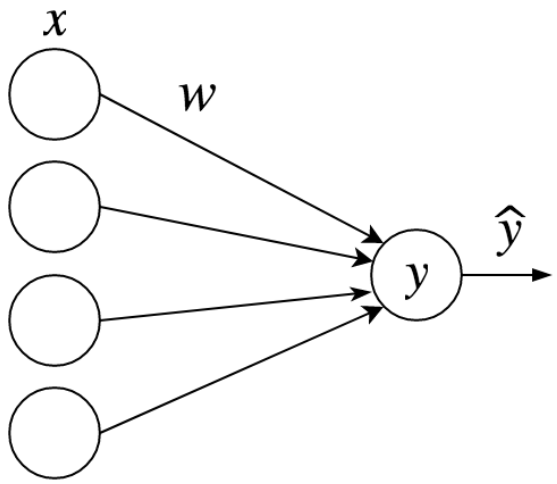


Figure 2.4: How several inputs x with an assigned weight w creates an output $\hat{y}$.

$$\hat{y} = \phi(x^{\intercal}W) \tag{2.10}$$

Each connection sends the output of a neuron i as an input to a neuron j and has a weight w_{ij}. Neuron i has an output $\hat{y}_i$ that depend on the neuron's input.

The propagation function (Equation 2.11) computes the input for neuron j from all connections in the previous layer in the network:

$$p_j = \sum_i \hat{y}_i \times w_{ij} \Rightarrow \hat{y} = \phi(p_j) \tag{2.11}$$

The weights in a network are all the parameters θ of the network that can be tuned, i.e., θ is the vector of all weights. If we adjust the weights w in the connections between two layers, not only is the output $\hat{y}$ affected, but also the neurons in the later layers that are connected to that neuron.

The goal of an ANN is to predict an output based on the network input data. Sending in data to a neural network with random weights will not produce any usable result. It is therefore necessary to update all connections to produce more accurate results. That is done by performing back-propagation for each training value:

1. Propagate forward to produce a prediction (network output)
2. Calculate the error E based on the prediction and the actual value
3. Propagate the neuron outputs backwards through the network to generate gradients for the weights to minimize the error
4. Apply the gradients to the weights in the network
5. (Optional) Update the weights in smaller steps. Subtract only a ratio α (learning rate) of the weights based on the gradients.

A common technique to better estimate the gradients for the network when updating its weights is averaging the gradient over multiple tranining examples, which are

referred to as *mini-batches*. By looking at a randomly selected training set of size x, often 128 or 256, back-propagation calculates the gradient using all training examples and applies the average gradient to the network [17]. Although this method is slower at first in terms of calculation time, it increases the prediction accuracy and will further improve as the batch-size increases by getting a less noisy estimate of the gradient. That way, it will handle noise in the training better, but also in most cases be a faster alternative than computing each individual training example.

3

Methods

This chapter will go through the the implementation of the data collection, the structure of the environment simulation and the algorithms used in this project. Here we will, with the theory presented in the last chapter, describe how to solve the problems presented in Chapter 1.

3.1 Structure of project

The project consists of two phases. The first phase is to focus on a theoretical investigation and outline proposals to solve our problems. We would have to create a hypothesis about what method would be the best solution for each use-case. This is to not put unnecessary work into a method that is not optimal or does not work at all. The second phase focuses on implementing one or more use-cases to verify theory vs. actual outcome in a simulation of a cloud environment. It does not mean that it is necessary to implement the solution to Ericsson's production system, but rather build a simulation and train an algorithm using real data from the vehicle ecosystem.

The RL algorithm requires certain information that must be built into the environment to function properly. For each IM in the system, the environment needs to know the usage of CPU, disk, memory, and network at any given time, as the reward function of the algorithm is dependent on these parameters. The algorithms used in this project will use that information to determine what actions are better at each state. These actions are constrained to how much memory each IM has available and how much of its computational resources it has left.

3.2 Implementation

This environment will be built with help from OpenAI Gym [18], which is a toolkit for developing and comparing RL algorithms. It is used to design the environment and compare the results from the different algorithms. This was done in Python 3.6 and the data collection, described in Section 3.3, was programmed in Java.

3.3 Collecting data

The resource information collected from each IM is the following:

- CPU usage
- Disk usage
- Memory usage
- Network usage
- VM type

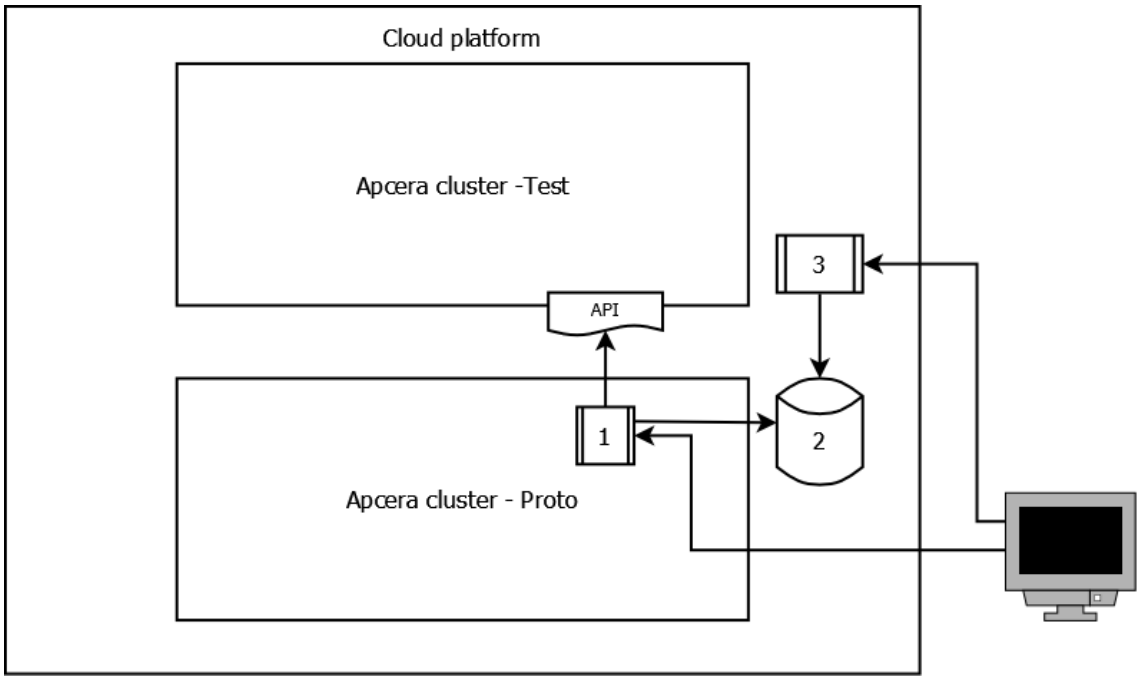


Figure 3.1: 1: Apcera job connected to the Test cluster API. 2: MySQL database. 3: Remote machine (AWS EC2 VM) running the algorithms.

Figure 3.1 visualizes the setup behind the data collection. We created an Apcera job running on the Proto cluster connected to the Test cluster's API. Because all clusters are online 24/7, the job can collect data without any interruptions. While running, it will store the data to the database, as seen in Appendix A, and can be used by the remote machine to train at a later time.

The reason why we connect to the Test cluster through the Proto cluster is mostly security, but also the potential performance impact of running the job on the real cluster itself. Because the Apcera job works in bursts every 60 seconds, connecting to the API and the database multiple times, we wanted to neglect that burst impact on the real cluster by deploying it to a different cluster.

3.4 Environment

The environment is built as a simulation of the Apcera system. An illustration of how Apcera works can be seen in Figure 3.2. When training, the action taken from state s_i is sent to the simulation and will then return the next state s_{i+1}. The effect of each action is built to work like the real system, e.g. adding an IM will result in more resources available for the cluster.

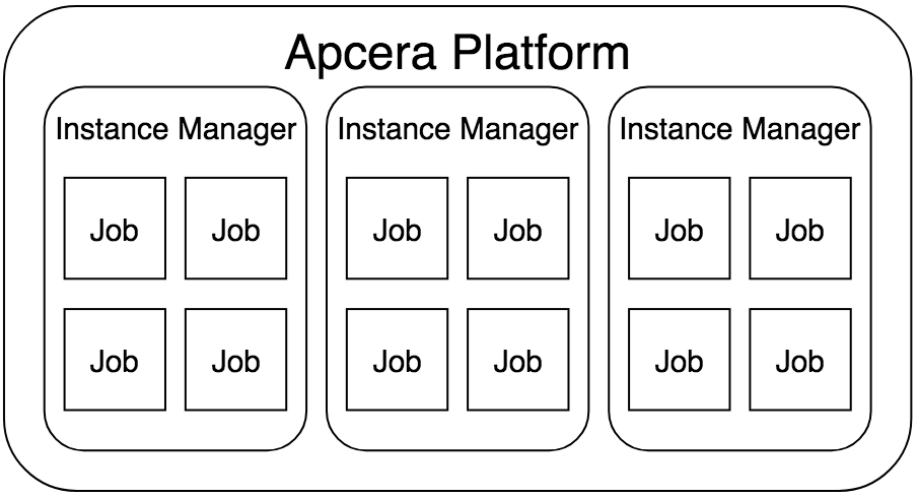


Figure 3.2: The Apcera platform. Each IM has several jobs running inside them.

Each IM has a specified size of each resource. CPU is a time resource measured in ms, and disk, memory and network are measured in GB. The size of these resources depends on what type of VM is rented from AWS. Certain VMs are memory optimized, CPU optimized or even all-round machines [2]. In this project, two types of VMs with different specifications and prices are used and the VM specifications can be seen in Table 3.1.

The number of types can be extended for a more complex problem solving. Obviously, increasing the number of VM types will increase the complexity of the problem and influence the performance of the algorithms. Therefore only two types are used in this project, as that is the minimum to showcase the usefulness of the algorithms.

VM Type	CPU	Disk	Memory	Network	Cost
i3.xlarge	4000 ms/s	475 GB	30.5 GiB	1 GB/s	0.214$/h
c5d.2xlarge	8000 ms/s	475 GB	16 GiB	1 GB/s	0.216$/h

Table 3.1: The different IM types and their available resources

Everything in the current and the following subsections are shared among all algorithms tested, i.e. all states, actions, rewards etc. in the MDP are the same. This is in order to keep the comparisons fair by retaining the pre-conditions.

3.4.1 States

The state variable of the MDP is defined as:

- $S = \{time, cpu, disk, mem, net, nr1, nr2\}$, where:
  - $time$ is the time of day. $0 \leq time \leq MaxTime$
  - cpu is the CPU usage of the system. $0 \leq cpu \leq 100$
  - $disk$ is the disk usage of the system. $0 \leq disk \leq 100$
  - mem is the memory usage of the system. $0 \leq mem \leq 100$
  - net is the network usage of the system. $0 \leq net \leq 100$
  - $nr1$ is the number of IMs of type 1. $0 \leq nr1 \leq 20$
  - $nr2$ is the number of IMs of type 2. $0 \leq nr2 \leq 20$

The variable $time$ is a looping variable that cycles through day, starting at 00:00 with a value of 0, later ending at 23:59 with a value of $MaxTime$. The $MaxTime$ variable is dependant on the interval t of each measurement:

$$MaxTime = \frac{24 \times 60 \times 60}{t} = \frac{86400}{t} \tag{3.1}$$

For example, if the interval t is 10 seconds, then $MaxTime = \frac{86400}{10} = 8640$. For this project, the interval is set to 600 seconds, or 10 minutes, which gives us $MaxTime = 144$. That means the variable $time$ can have 144 different values.

The result of all variables is a model where the state space has $144 \times 101^4 \times 20^2 = 58 \times 10^{11}$ unique states. It is possible to shrink the state space at the cost of precision, e.g. halve the precision of the usage or increase the interval t.

3.4.2 Actions

In an MDP environment $\{S, A, P_a(s, s'), R_a(s, s'), \gamma\}$, an action from the action space A changes the state s_i to the next state s_{i+1}.

The size of the action space A is linear to the number of VM types that are used. There are two actions available for each VM type: remove and add a VM. Apart from these actions, there is always an option to do nothing. This results in up to $2 \times x + 1$ available actions, where x is the number of VM types. For this environment setup with two VM types, we have $2 \times 2 + 1 = 5$ actions. The amount of actions is directly tied to the number of VM types used in the model and will require alterations in the models state space for compatibility. The actions used in this project with two VM types are listed as the following:

- a_1: Add a VM of type 1
- a_2: Add a VM of type 2
- a_3: Remove a VM of type 1
- a_4: Remove a VM of type 2

- a_5: Do nothing

The algorithm's job is to determine which is the best action in each given state, and it has to choose from one of the actions mentioned above. In Section 3.4.3, it is explained that the actions $a_1 - a_4$ should have an immediate, negative reward to avoid alternating between adding and removing VMs. All actions except the "Do nothing" action are penalized.

3.4.3 Reward function

The reward function is what decides what actions are *good* and *bad* in every state. A higher reward is better. This function will guide the MDP through states with the least amount of negative rewards, so a lot of focus is put on this part of the environment.

The calculated expected reward $E[R]$ is the same value function discussed in Section 2.1.2 and allows the algorithms to predict the future discounted rewards of all states:

$$E[R] = E\left(\sum_{t=0}^{\infty} \gamma^t R_{\pi(s_t)}(s_t, s_{t+1})\right) \tag{3.2}$$

where the discount value γ is 0.95. As mentioned in Section **??**, so γ should have a lower value than 1.

The reward at each state taking action a is defined in Equation 3.3:

$$R_{\pi(s_t)}(s, s') = (\beta \times C(a, s') + (1 - \beta) \times P(s')) \times P_{mult} \tag{3.3}$$

$$C(a, s') = c_{frac} \times a + c_{vm} \times u_{s'} \times t \tag{3.4}$$

$$P(s') = \frac{Pc}{3600} \times t \times \begin{cases} \left(1 + \dfrac{u - P_{limit}}{P_{limit}}\right) & \forall u \in U, \text{ if } u > P_{limit} \\ 0 & \text{else} \end{cases} \tag{3.5}$$

$$P_{mult} = \begin{cases} 5 & \text{if } \exists u \in U \text{ where } 100 \leq u, \text{ or } u_{s'} < 3 \\ 1 & \text{else} \end{cases} \tag{3.6}$$

where:

- π is an arbitrary policy.
- β is a balance variable that acts like a weight between the cost and the penalty rewards.
- $C(a, s')$ is a cost function of running the cluster and adding/removing IMs.
- $P(s')$ is a penalty function (negative reward) for situations when the resource usage is too high. It help the algorithm to avoid scenarios where a resource usage exceeds a certain limit. When this penalty is activated depends on P_{limit}.

- $P_{mult} = 5$ is a penalty multiplier. We multiply all costs and penalties with a factor of 5 if: the usage of any resource reaches 100%, or if there are less than three IMs available, as three IMs is the recommended minimum number of IMs in production stated in the Apcera documentation [19].
- $c_{frac} = \frac{c_{vm}}{3600} \times t \times 5$ is an immediate penalty for adding or removing a VM. The penalty is the cost of running a VM for 5 steps.
- a is the number of IMs removed or added to the cluster, as described in Section 3.4.2
- c_{vm} is the cost of running a VM. The prices are specified in Table 3.1
- $u_{s'}$ is the number of IMs being online at state s'. That means, after action a at state s has been taken, we have u IMs online at the next state.
- t is the time interval between steps specified in Section 3.4.1.
- Pc is the penalty cost of using too much resources from the cluster. We used $Pc = 5\$/h$.
- $u \in U$ represents the usage of each resource in the cluster.
- P_{limit} is a limit on when a penalty should be added for using more resources than said limit. We used $P_{limit} = 0.7$, and when a resource in U exceeds the limit, an increasing penalty is added.

The goal and reasoning behind these penalty functions are the following:

- Lower the cost of running the cluster.
- Always have enough extra resources if an IM crashes or disconnects from the cluster.
- Not have the resources reach or exceed 100%.

The reason for the actions having an extra immediate cost in $C(a, s')$ is to penalize excessive switching between adding and removing VMs. This will in turn encourage the algorithm to choose action 5 and do nothing.

3.5 Simulation

As stated in Section 3.1, a simulation of the Apcera platform was built for the environment to train with instead of a real system. Every action taken by the algorithms will affect the resources available in the simulations, and the first thing the simulation does when starting an algorithm is to connect to the database and load the following values:

- The current number of IMs being online
- The resources available for each VM type
- The total resource usage of the cluster

The number of IMs being online is a variable that the algorithms have direct control over by performing a certain action. Four of the actions described in Section 3.4.2 will either increase or decrease the number of IMs, while the last action does

nothing. For each episode in the simulation, the number of IMs is randomized to help the algorithm to experience different configurations and be able to adapt to any situation.

Furthermore, the resources available for each VM type are static values that cannot be affected by the algorithm. It is only used to calculate the usage percentage based on how many IMs are online and how much resources is actually used in the cluster.

Finally, the total resource usage is another value the algorithm cannot change, but will instead change on its own between any two steps. For when initializing the simulation, it loads all usage values with an interval of t seconds from the database into an array. E.g. in Figure 3.3 which show some test data collected from an already existing cloud, we can see the memory usage varies between 335 GB and 200 GB during the day. After each taken step, the simulation traverses through these values until the training episode stops. This applies for all resources described in Section 3.3.

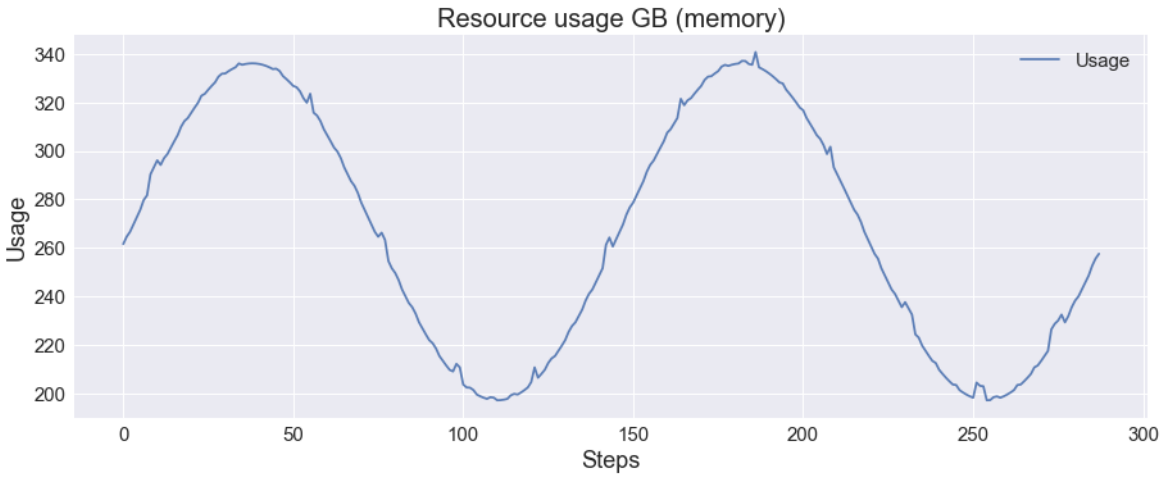


Figure 3.3: 288 samples (two days) of the normal memory resource usage in the cloud cluster.

3.6 Algorithms

There are many different types of machine learning algorithms that can solve different kinds of problems. For this project, without any annotated data to train with, we chose to focus on RL algorithms that can instead rely on a reward function to determine how *good* the current state is.

In the case of algorithms such as Q-Learning and SARSA that we wanted to explore, there are limitations on how big a problem can be. Both these methods were unsuitable to solve the problem at hand due to the size of the problem. The two tables we need to solve our problem, a state value table $V(s)$ and a q-value approximation table $Q(s, a)$, are considerably large with the state space used for this problem.

The state value function $V(s)$ would require an array with 7 dimensions with a size of $[101, 101, 101, 101, 144, 20, 20]$, and an action-value function $Q(s, a)$ of 8 dimensions

with a size of $[V(s), 5]$. These functions where introduced in Section 2.1.2 and 2.1.3. In Section 3.4.1, we calculated that there are 58×10^{11} unique states in S, which results in 29×10^{12} unique state-action pairs. Keeping track of so many different float values is not viable. Therefore, there will not be any results of the performance for Q-Learning in Chapter 4.

However, we can still implement Q-Learning and SARSA with a neural network (NN) as an alternative to a table of arrays, and instead approximate the q-value with the network's output nodes.

For all the following algorithms implemented, a mini-batch size of 128 was used. It means that after 128 training samples have been collected, the algorithms can start to train the neural network.

3.6.1 Deep Q-Learning

Deep Q-Learning (DQL) follows the same principles as the Q-Learning algorithm. The difference is, instead of using an array to store all rewards, it approximates the optimal policy using a neural network as an equivalent of $Q(s, a)$ from Section 2.1.3. The layers in the network are the following for all algorithms in this project:

- Input layer: 7 neurons, one for each state variable described in Section 3.4.1
- Hidden layer: fully connected with 100 neurons
- Hidden layer: fully connected with 60 neurons
- Output layer: 5 neurons, one for each action described in Section 3.4.2

There are no tables for the action or state reward in this algorithm, i.e., the functions $Q(s, a)$ and $V(s)$. Instead we get the q-values from the output layer of the neural network by inserting the state as an input. With the network parameters θ, we denote the q-value by writing $Q(s, a; \theta)$. By inserting an input state s into the network, we get the output q-values for all actions.

However, we have to update the neural network's parameters θ in order to better choose the appropriate action to take at each state. We do that by calculating the target value, also called temporal difference (TD-target) Y_t as seen in Equation 3.7.

$$Y_t^{DQN} \equiv R_{t+1} + \gamma \max_a Q(s_{t+1}, a; \theta_t) \tag{3.7}$$

To update the parameters of the network, we use an adaptive learning approach called RMSprop, proposed by Geoff Hinton [20] [21]. In Equation 3.8, we update the parameters of θ_t with its old parameters towards the target Y_t. It divides the learning rate α by an exponentially decaying average of squared gradients $E[Y^2]$ [20]. $E[Y^2]$ in Equation 3.10 is a mean square error function of the TD-target Y_t.

$$\theta_t = \theta_t + \Delta\theta_t \tag{3.8}$$

$$\Delta\theta_t = -\frac{\alpha}{\sqrt{E[Y^2]_t + \epsilon}} Y_t \tag{3.9}$$

$$E[Y^2]_t = \gamma E[Y^2]_{t-1} + (1-\gamma)Y_t^2 \tag{3.10}$$

where:

- $\alpha = 0.0001$
- $\gamma = 0.95$
- $\epsilon = 0$ (Not the same ϵ for selecting random actions)

The appropriate parameter changes in θ, in Equation 3.9, are calculated using back-propagation [22] and is handled by Keras in Python [21].

Algorithm 2 for DQL is performed for a number M of episodes. When there are at least 1000 transitions in the replay memory D, it will start to randomize mini-batches of these transitions and train the network.

Algorithm 2 Deep Q-Learning

Initialize replay memory D
Initialize network parameters θ
for episode = 1,M **do**
    Reset environment ϕ
    Reset state: $s_1 \leftarrow \phi_1$
    for t = 1, T **do**
        Get policy: $\pi \leftarrow \underset{a}{\operatorname{argmax}}\, Q(s_t, a; \theta)$
        Select action a_t from π with a probability of ϵ
        else select random action
        Perform action a_t
        Get next state/reward/terminal: $s_{t+1}, r_{t+1}, ter, \leftarrow \phi(s_t, a_t)$
        Store transition $D \leftarrow (s_t, a_t, r_{t+1}, s_{t+1}, ter)$
        if 1000 < D **then**
            Sample minibatch of transitions $(s_j, a_j, r_{j+1}, s_{j+1})$ from D
            $\mathrm{Y}_t = \begin{cases} r_{j+1} & \text{if episode terminates at } s_{j+1} \\ r_{j+1} + \gamma \max\limits_a Q(S_{j+1}, a; \theta) & \text{else} \end{cases}$
            $\theta \leftarrow \theta + \Delta\theta$ from Equation 3.9 using Y_t
        end if
    end for
end for

3.6.2 Double Deep Q-learning

The advantage of using DDQL over DQL is that it avoids overestimating the reward when calculating the TD-target [10]. To achieve this, DDQL uses an additional neural network so that it has both an online and an offline network, with the parameters θ and θ^- respectively. The target value is calculated from the offline network using the best action according to the online network, as seen in Equation 3.11. The target value is then applied on the online network as in Equation 3.8 using RMSprop. Every time an update is applied on the online network, the networks switch places

with a probability of 50%. The algorithm is very similar to DQL and can be seen in Algorithm 3.

$$Y_t^{DDQL} \equiv R_{t+1} + \gamma Q(S_{t+1}, \underset{a}{\operatorname{argmax}}\, Q(S_{t+1}, a; \theta_t); \theta_t^-) \tag{3.11}$$

Algorithm 3 Double Deep Q-Learning

Initialize replay memory D
Initialize online network parameters θ
Initialize offline network parameters θ^-
for episode = 1,M **do**
    Reset environment ϕ
    Reset state: $s_1 \leftarrow \phi_1$
    for t = 1, T **do**
        Get policy: $\pi \leftarrow \underset{a}{\operatorname{argmax}}\, Q(s_t, a; \theta)$
        Select action a_t from π with a probability of ϵ
        else select random action
        Perform action a_t
        Get next state/reward/terminal: $s_{t+1}, r_{t+1}, ter \leftarrow \phi(s_t, a_t)$
        Store transition $D \leftarrow (s_t, a_t, r_{t+1}, s_{t+1}, ter)$
        if 1000 < D **then**
            Sample minibatch of transitions $(s_j, a_j, r_{j+1}, s_{j+1})$ from D
            $\mathrm{Y}_t = \begin{cases} r_{j+1} & \text{if episode terminates at } s_{j+1} \\ r_{j+1} + \gamma Q(S_{j+1}, \underset{a}{\operatorname{argmax}}\, Q(S_{j+1}, a; \theta); \theta^-) & \text{else} \end{cases}$
            $\theta \leftarrow \theta + \Delta\theta$ from Equation 3.9 using Y_t
            Randomly switch θ and θ^-
        end if
    end for
end for

3.6.3 SARSA

SARSA (State–action–reward–state–action) [23] is very similar to Q-learning, but in its algorithm loop, as seen in Algorithm 4, it predicts the next action after it has already taken a step, hence the last "action" in the name.

Unlike Q-learning, SARSA's target value, in Equation 3.12, does not estimate the future reward like DQL and DDQL do. Instead, it estimates the reward from the actual action taken in state s_{t+1} using an *epsilon-greedy* policy while training. That makes it learn from actual experience and actions the NN choose.

$$Y_t^{SARSA} \equiv R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}; \theta_t) \tag{3.12}$$

Algorithm 4 SARSA

Initialize replay memory D
Initialize network parameters θ
for episode = 1,M **do**
    Reset environment ϕ
    Reset state: $s_1 \leftarrow \phi_1$
    Get policy: $\pi \leftarrow \underset{a}{\operatorname{argmax}}\, Q(s_t, a; \theta)$
    Select action a_1 from π with a probability of ϵ
    else select random action
    for t = 1, T **do**
        Perform action a_t
        Get next state/reward/terminal: $s_{t+1}, r_{t+1}, ter, \leftarrow \phi(s_t, a_t)$
        Get policy: $\pi \leftarrow \underset{a}{\operatorname{argmax}}\, Q(s_{t+1}, a; \theta)$
        Select action a_{t+1} from π with a probability of ϵ
        else select random action
        Store transition $D \leftarrow (s_t, a_t, r_{t+1}, s_{t+1}, a_{t+1}, ter)$
        if 1000 < D **then**
            Sample minibatch of transitions $(s_j, a_j, r_{j+1}, s_{j+1}, a_{j+1})$ from D
            $\mathrm{Y}_t = \begin{cases} r_{j+1} & \text{if episode terminates at } s_{j+1} \\ r_{j+1} + \gamma Q(s_{j+1}, a_{j+1}; \theta) & \text{else} \end{cases}$
            $\theta \leftarrow \theta + \Delta\theta$ from Equation 3.9 using Y_t
        end if
    end for
end for

3.6.4 AC3

Unlike the other algorithms, Asynchronous Actor-Critic Agents (AC3) can utilize multiple or all cores in a CPU making it multi-threaded [11] (asynchronous), but it also uses the same neural network for both action decision and future value estimation (actor-critic).

The NN has two output layers, one for the policy estimation $Q(s, a)$ (the actor), and one for value estimation $V(s)$ (the critic). The rest of the network is shared, so when the network is updated with gradients, both outputs are affected.

At the start, the AC3 algorithm creates multiple agents, one for each CPU core with their own NN and environment. How each agent behaves can be seen in Algorithm 5, and the structure can be seen in Figure 3.4. At the start, they explore the environments separately, and every n:th step they update the global network with the gradients made from their own local network. They also update their local networks parameters from the global network periodically to stay updated.

Algorithm 5 AC3 - pseudocode for each actor

1: //Assume global shared θ, and local θ^-
2: Initialize local network parameters from the global network $\theta^- \leftarrow \theta$
3: Initialize network gradients $\Delta\theta \leftarrow 0$
4: **for** episode = 1,M **do**
5: Reset environment ϕ
6: Reset state: $s_1 \leftarrow \phi_1$
7: **for** t = 1, T **do**
8: Get policy: $\pi \leftarrow \underset{a}{\operatorname{argmax}}\, Q(s_t, a; \theta)$
9: Select action a_t from π with a probability of ϵ
10: else select random action
11: Perform action a_t
12: Get next state/reward/terminal: $s_{t+1}, r_{t+1}, ter, \leftarrow \phi(s_t, a_t)$
13: $\mathrm{Y}_t = \begin{cases} r_{i+1} & \text{if episode terminates at } s_{j+1} \\ r_{i+1} + \gamma \max\limits_{a'} Q(s_{i+1}, a'; \theta^-) & \text{else} \end{cases}$
14: $\Delta\theta \leftarrow \Delta\theta + \Delta\theta_t$ from Equation 3.9 using Y_t
15: **if** t mod $I_{target} == 0$ **then**
16: Update target network $\theta^- \leftarrow \theta$
17: **end if**
18: **if** t mod $I_{AsyncUpdate} == 0$ or s_t is terminal **then**
19: Update global network parameters $\theta \leftarrow \theta + \Delta\theta$
20: Clear gradients $\Delta\theta \leftarrow 0$
21: **end if**
22: **end for**
23: **end for**

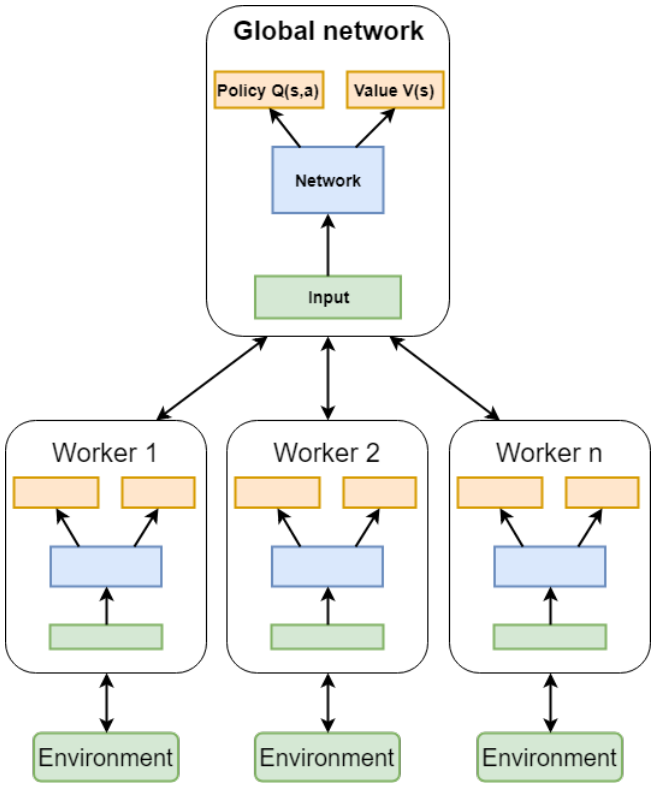


Figure 3.4: AC3 structure. Multiple agents train simultaneously on their separate environment to train the global network.

4

Results

This chapter will present the result of the algorithms and their performances during the training, and how it handles auto-scaling after training for 1000 episodes.

Unfortunately, no results from AC3 were usable. One possible reason is the shared network for both the policy and value output in the NN. Applying one gradient to minimize the error of the policy estimation will affect the value output, and vice versa. The end result is always a policy that chooses action 1 from Section 3.4.2, i.e. it will always try to add a VM at each time step. It will stop at 20 VMs, as the max number in the cloud, and then stay there forever.

4.1 Training results

Analyzing the training results can seem counterproductive as it cannot give any insight as to how each algorithm behaves. However, the results during training can still explain as to *why* one algorithm performs better than the others.

4.1.1 Episode reward

The episode rewards shows how *good* the algorithms perform during training. The results in Appendix B shows how each algorithm explores new paths and solutions for 1000 episodes.

As discussed in Section 2.1.3, we used an ϵ-greedy policy with ϵ decreasing after each episode. While ϵ decreases, so does the number of random actions taken, and the algorithms instead choose actions to maximize the reward. This results in the reward variations gradually shrink, but the overall rewards increase as the training progresses.

As seen in Appendix B, the rewards for SARSA are more concentrated than for both DQL and DDQL, but the graph also shows that the rewards contain more prominent spikes. The higher reward concentration results in the mean rewards being lower for SARSA than the rest of the algorithms shown in Table 4.1. These rewards are the total returns from the reward function after 288 steps. Column 1 shows the mean reward for all episodes, and column 2 is the the reward after 1000 episodes of training. The last column is from running a simulation of two days with 288 steps without any random actions.

Even though DDQL has the worst mean reward during training, it still ends up with the best reward in the end of the training and the simulation while both DQL and SARSA fall behind.

Algorithm	Training Mean Reward	Training End Reward	Simulation Reward
DQL	-142.55	-87.99	-86.43
DDQL	-143.26	-86.93	-84.31
SARSA	-123.78	-87.87	-85.02

Table 4.1: The mean and end reward after training for 1000 episodes, as well as the reward for running it through a simulation. Results from Appendix B. Higher is better.

The simulation reward reflects how well the algorithms perform, and from the table above we can see that DDQL is the best. We can alter the value of several variables in the environment mentioned in Section 3.4 to change the desired behavior of the algorithms, e.g., how close the resource limit we should be, or how conservative the algorithms should be with adding or removing IMs. For all combinations of values assigned to each variable, DDQL will follow the altered reward function better than DQL and SARSA.

4.1.2 Episode cost

Compared to episode rewards, episode cost only takes the cost of running the cloud cluster into consideration, i.e. paying for the VMs that are online.

The results from the episode cost correlate with the pricing of running the cluster, so if the cost during training is high, so will the cost in the simulation be. The cost after training will be shown later in Section 4.2.3.

In Table 4.2, we can see that, even though the training reward for SARSA is better than DQL, the pricing differs by quite a lot, especially the mean cost. SARSA falls behind on both the mean and end cost, while DDQL's performance is better.

Algorithm	Mean Cost	End Cost
DQL	0.591	0.525
DDQL	0.550	0.507
SARSA	0.688	0.608

Table 4.2: The mean cost and end reward after training for 1000 episodes. Results from Appendix C. Lower is better.

The weight between the cost and penalties can be altered in the environment to encourage other behaviours, e.g. avoiding reaching a certain resource usage limit, or lowering the cost.

4.2 Comparisons

To make a proper comparison between the algorithms used in Section 3.6, we need to define what abilities and attributes are desired for the cloud infrastructure.

The first attribute that will be compared is the speed of the algorithms, i.e. how long it takes to train for 1000 episodes. Since more training will give better result, speed will affect the outcome of the choices made by each algorithm.

Another interesting aspect of each algorithm is how much resources it uses in percentage during a simulation. In the training, a penalty will initiate when a resource usage reaches 70% and will grow linear with the usage. Lastly, the cost of running the IM cluster with each algorithm will be compared. This is the most important aspect of the end result.

All of these results will be compared after each algorithm has completed 1000 episodes of training. They can all be seen in Appendix D, E, F, and G.

4.2.1 Speed

The computation time for each algorithm can be seen in Table 4.3, and as shown, DQL and SARSA have very similar results but DDQL takes much longer to complete. This is a consequence of the two neural networks DDQL uses. When calculating the TD-target, DDQL has to predict the output for two neural networks instead of one. This doubles the prediction time, for predicting 128 training samples, from $0.5ms$ to $1ms$ for each training step for DDQL compared to DQL and SARSA.

Algorithm	Time (s)	% faster than DDQL
DQL	$953s$	28.0%
DDQL	$1323s$	—
SARSA	$971s$	26.6%

Table 4.3: The time it takes for each algorithm to complete 1000 episodes of training. DDQL is the baseline for the speed comparison.

As the complexity of the NN increases, the more noticeable the time differences will be, as the output prediction for each state s will take longer.

4.2.2 Stability

In Section 3.4.3 we introduced a limit variable to avoid reaching a certain resource usage. It is not a hard limit, but after reaching 70% usage a penalty activates and grows linearly with the usage of each resource.

The resource limit is made to encourage resource overhead in case any VM or datacenter goes down. There are always other VMs that can take over the processes from the VMs that disappeared. This is to avoid slowdown or instabilities in the cluster.

Reaching a resource usage over 90% is not considered stable and should be avoided. In terms of stability, SARSA performs best here and does not exceed 75% resource usage at any given time in the simulation. Although SARSA is more stable than both DQL and DDQL, its stability results in a higher cost as presented later in Section 4.2.3.

Algorithm	Mean	Over 70%	Over 75%	Over 80%	Over 85%	Over 90%
DQL (CPU)	68.8%	51%	45%	39%	29%	9%
DDQL (CPU)	76.8%	76%	58%	42%	20%	0%
SARSA (CPU)	62.0%	17%	0%	0%	0%	0%
DQL (Mem)	62.8%	37%	4%	0%	0%	0%
DDQL (Mem)	76.5%	78%	58%	38%	25%	0%
SARSA (Mem)	65.6%	19%	0%	0%	0%	0%

Table 4.4: How much resources are used during a simulation with multiple limits to compare. Results from Appendix E.

4.2.3 Pricing

In terms of cost, Table 4.5, just like in Table 4.2, shows that DDQL is better than both DQL and SARSA. That is because, as stated in Section 4.2.2, DDQL's resource usage stays closer to the resource limit.

To get a sense on how much cheaper each algorithm is compared to a cluster without any auto-scaling, we use a baseline cost of 0.6$ per step, the maximum cost of DDQL per step from Table 4.5, to calculate the total cost of running the cluster for two days. The cost for this scenario adds up to 173$ and the resource usage would instead look like Figure 4.1.

Algorithm	Cost	Max cost/step	Mean cost/step	Savings $	Savings %
Base	**173$**	**0.60$**	**0.60$**	**-**	**-**
DQL	167$	0.63$	0,58$	11$	3.5%
DDQL	142$	0.60$	0,49$	31$	17.9%
SARSA	171$	0.71$	0,59$	2$	1.2%

Table 4.5: The cost of running the cluster for two days with decisions made from each algorithm. Results from Appendix G.

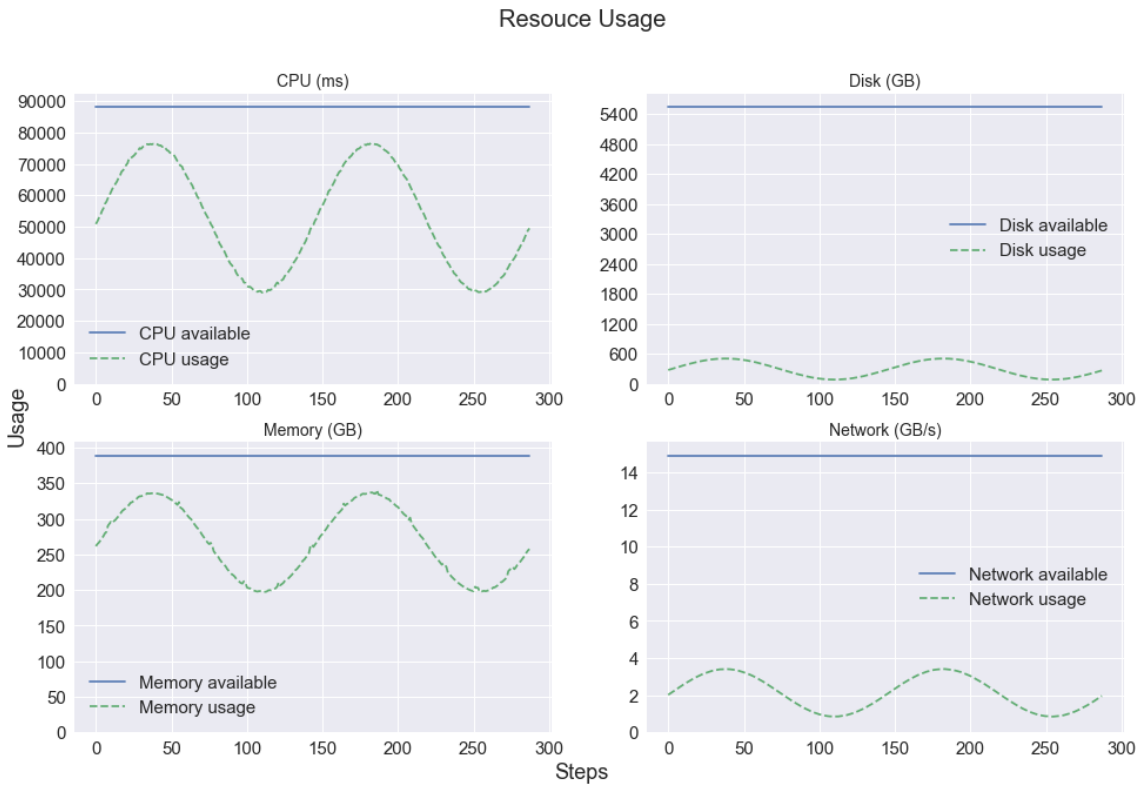


Figure 4.1: The resource usage on cluster without auto-scaling.

5

Discussion

This chapter will discuss and evaluate the results from Chapter 4, suggest changes to the current solution as well as possible future work.

5.1 Evaluating results

In this section, we will evaluate the resource usage from each algorithm and its stability. These evaluations will focus on the results from Appendix D and E, as presented in Section 4.2.2.

5.1.1 CPU/Memory

The algorithms only needed to keep track of the CPU and memory usage. We can see in Appendix F that they choose to use more 'i3.xlarge' VMs than 'c5d.2xlarge', as the cluster generally needs more memory than CPU resources.

This reflects the resource usage percentage of the cluster, shown in Appendix E, and the load on both resources only differ by a few percentages. This behaviour is caused by the reward function discouraging the cluster to reach 70% usage or higher of any resource. The result is that the algorithm will try to balance the load between the different resources, as that is the most cost effective alternative.

5.1.2 Disk/Network

As seen in Appendix D, both the disk and network usage data collected from the cluster were so low that they had no impact on the algorithms choices. There is no instance where these resources reaches 70% usage or higher. Therefore, the VMs used to keep the CPU and memory usage down, will create an excess amount of disk and network resources available.

5.1.3 Stability

The stability of the cluster depends on how many VMs that are online and the current resource load. The different algorithms will react VM and reacts , and is decided on the resource limit.

The limit can be set to any value or function type. The function type will affect how harsh the penalties are when reaching a certain resource usage.

The kind of changes we can do to the limit function are: adjust the resource limit or limit cost, make the function exponential or linear, remove the penalty, or implement a hard limit to completely avoid reaching the limit. This function will have the most impact on the reliability of the system and how the algorithms will behave after training.

The results in Appendix D shows an example of a soft limit where it is possible for the cluster to exceed the limit, given that the resource usage will go down eventually.

5.2 Algorithm efficiency

One thing to take into consideration is that the potential savings made by each algorithm are linked to how the clusters usage fluctuates. If the usage is mostly static from day to night, then there are no real improvements to be made if we already have an optimal configuration of IMs. However, these algorithms can still help decide how to create these static configurations.

With or without fluctuations in resources, the algorithms can configure the cluster better than any human ever could, given that the problem is complex enough. Using one or possibly two VM types, the cluster could be configured manually by a human, but it becomes increasingly harder to consider all parts, four resources plus the cost, with even more VM types to reach maximum efficiency.

5.3 Future work

Further development can introduce features such as:

- Instance placement. Determine in what VM type to place a job and its instances.
- Scale the cluster for each VM type individually instead of all types.
- Determine between two VM types which one is better for a specific cloud configuration and usage.
- Simulate random VM or data-center crashes.

Instance placement in conjunction with scaling each VM type individually can improve the cloud optimization even further, potentially making it more stable in the process by avoiding overloading certain IMs. As stated in Section 1.4, there is no functionality at the moment to make an active choice of in which IM to put a job instance with Apcera. This is simply a limitation in Apcera's functionality.

5.3.1 Neural network improvements

There are several alternatives to improve or adjust the current solution in this project. Since this project did not focus on finding the best NN model, there is potential for improvements in training speed and/or model performance. Neural networks are a great tool for problems solving, but what truly makes them great is when they are designed to specifically handle a certain problem. That requires more time and effort than what was put into this project and is therefore an aspect that could be improved upon.

The NN model created for this project was sufficient to give proper results. Although, if more VM types were to be introduced, it can become harder for the NN to predict the right action. The complexity of the NN might have to be increased which will require more training time.

6

Conclusion

Solving a problem of this size requires efficiency and scalability that certain methods are not viable anymore. In this project, we focused on algorithms that either already had a neural network in its design, or altered an already existing method to include one, such as Q-Learning and SARSA. Without it, the computational power required would have been too high.

After training three algorithms for 1000 episodes, we observe several differences between them. While DDQL is the more time consuming algorithm tested in this project, it makes up for it in performance while training and the potential cost reduction of running the cloud cluster. The somewhat higher training reward indicates that DDQL will adjust better to any changes made to the reward function in Section 3.4.3, and as a result function better than both DQL and SARSA.

Worst of is SARSA which only marginally improves the cost by 1.2%, but it still raises the overhead for resources more efficiently during the simulation than without any auto-scaling. SARSA is therefore considered the most stable alternative. While DQL is cheaper to use, it is still the most unstable algorithm to perform auto-scaling with. It also only improves the cost by 3.5%. DDQL is best in terms of optimizing the price and lowers the cost by 17.9% according to the simulations made in this project.

These possible saving is enough to consider this solution viable and can possibly be improved further by optimizing the neural network used. Not only is it saving money but also an automated solution, removing the need for human input to keep the cluster optimized.

Bibliography

[1] Docker. Accessed: 2018-08-10. [Online]. Available: https://www.docker.com/resources/what-container

[2] Aws. Accessed: 2018-01-21. [Online]. Available: https://aws.amazon.com/

[3] V. Chandola, A. Banerjee, and V. Kumar, "Anomaly detection: A survey," *ACM Computing Surveys*, vol. 41, no. 3, pp. 15:1–15:58, Jul. 2009. [Online]. Available: http://doi.acm.org/10.1145/1541880.1541882

[4] P. Garcia-Teodoro, J. E. Diaz-Verdejo, G. Maciá-Fernández, and E. Vázquez, "Anomaly-based network intrusion detection: Techniques, systems and challenges," *Computers Security*, vol. 28, no. 1, pp. 18 – 28, 2009. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167404808000692

[5] X. Dutreilh, S. Kirgizov, O. Melekhova, J. Malenfant, N. Rivierre, and I. Truck, "Using reinforcement learning for autonomic resource allocation in clouds: towards a fully automated workflow," in *7th International Conference on Autonomic and Autonomous Systems (ICAS'2011)*, Venice, Italy, May 2011, pp. 67–74. [Online]. Available: https://hal-univ-paris8.archives-ouvertes.fr/hal-01122123

[6] E. Barrett, E. Howley, and J. Duggan, "Applying reinforcement learning towards automating resource allocation and application scalability in the cloud," *Concurrency and Computation: Practice and Experience*, vol. 25, no. 12, pp. 1656–1674, 2013. [Online]. Available: http://dx.doi.org/10.1002/cpe.2864

[7] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, p. 436, 2015.

[8] C. J. C. H. Watkins and P. Dayan, "Q-learning," *Machine Learning*, vol. 8, no. 3, pp. 279–292, May 1992. [Online]. Available: https://doi.org/10.1007/BF00992698

[9] G. A. Rummery and M. Niranjan, "On-line Q-learning using connectionist systems," Cambridge University Engineering Department, CUED/F-INFENG/TR 166, September 1994. [Online]. Available: ftp://svr-ftp.eng.cam.ac.uk/reports/rummery_tr166.ps.Z

[10] H. van Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double q-learning," *CoRR*, vol. abs/1509.06461, 2015. [Online]. Available: http://arxiv.org/abs/1509.06461

[11] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. P. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning," *CoRR*, vol. abs/1602.01783, 2016. [Online]. Available: http://arxiv.org/abs/1602.01783

[12] Kubernetes nodeselector. Accessed: 2018-03-14. [Online]. Available: https://kubernetes.io/docs/concepts/configuration/assign-pod-node/

[13] M. van Otterlo and M. Wiering, *Reinforcement Learning and Markov Decision Processes*. Springer Berlin Heidelberg, 2012, pp. 3–42. [Online]. Available: https://doi.org/10.1007/978-3-642-27645-3_1

[14] L. P. Kaelbling, M. L. Littman, and A. W. Moore, "Reinforcement learning: A survey," *Journal of artificial intelligence research*, vol. 4, pp. 237–285, 1996. [Online]. Available: http://arxiv.org/abs/cs.AI/9605103

[15] C. J. C. H. Watkins, "Learning from delayed rewards," Ph.D. dissertation, King's College, Cambridge, 1989.

[16] S. M. Ross, *Introduction to stochastic dynamic programming*. Academic press, 2014.

[17] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," *arXiv preprint arXiv:1502.03167*, 2015.

[18] Openai. Accessed: 2018-03-19. [Online]. Available: https://gym.openai.com/docs/

[19] Apcera. Accessed: 2018-01-21. [Online]. Available: https://docs.apcera.com/

[20] S. Ruder, "An overview of gradient descent optimization algorithms," *CoRR*, vol. abs/1609.04747, 2016. [Online]. Available: http://arxiv.org/abs/1609.04747

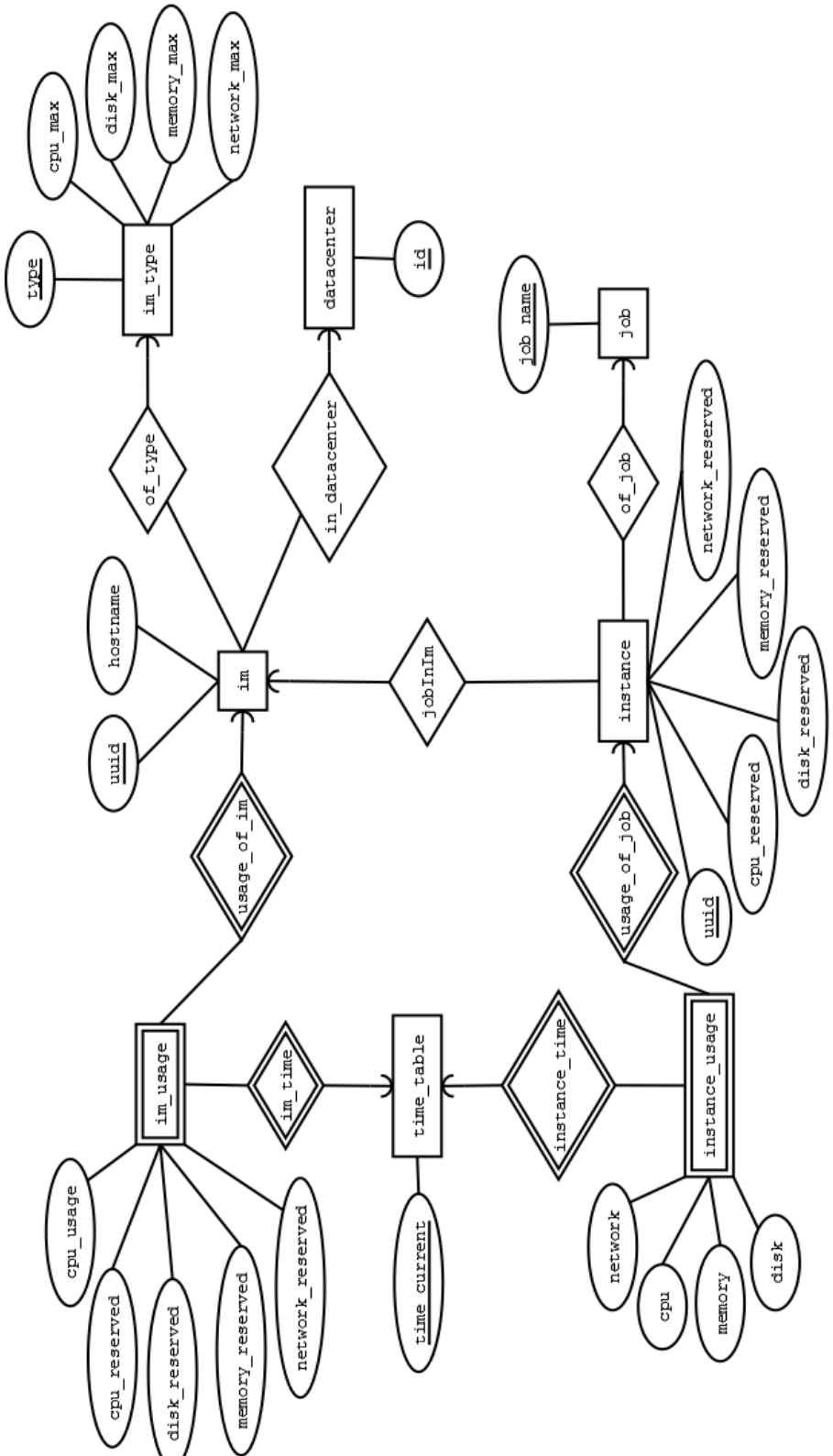
[21] "Keras optimizers," https://keras.io/optimizers/, accessed: 2018-05-28.

[22] P. J. Werbos, "Backpropagation through time: what it does and how to do it," *Proceedings of the IEEE*, vol. 78, no. 10, pp. 1550–1560, 1990.

[23] J. Gläscher, N. Daw, P. Dayan, and J. P. O'Doherty, "States versus rewards: dissociable neural prediction error signals underlying model-based and model-free reinforcement learning," *Neuron*, vol. 66, no. 4, pp. 585–595, 2010.

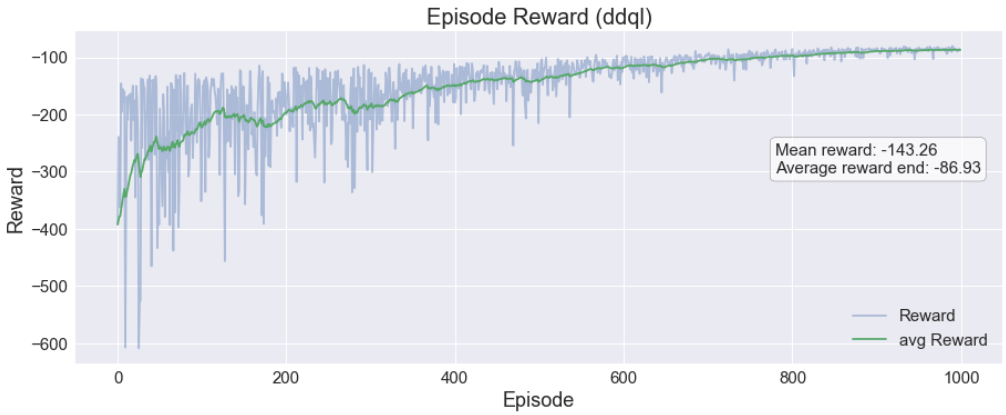
A

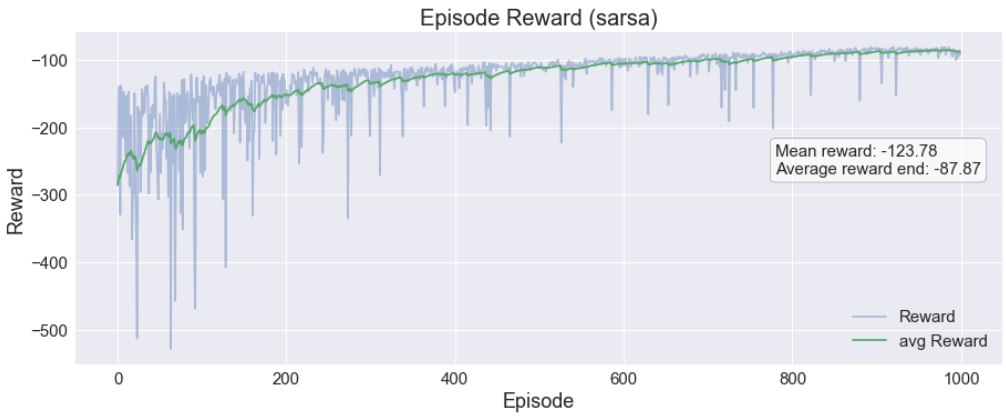
MySQL Database



B

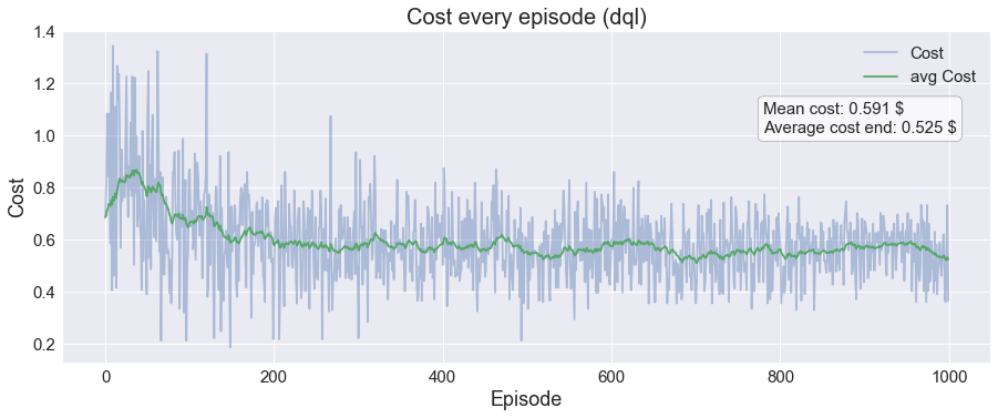
Training episode reward





C

Training episode cost



D

Cluster resource usage

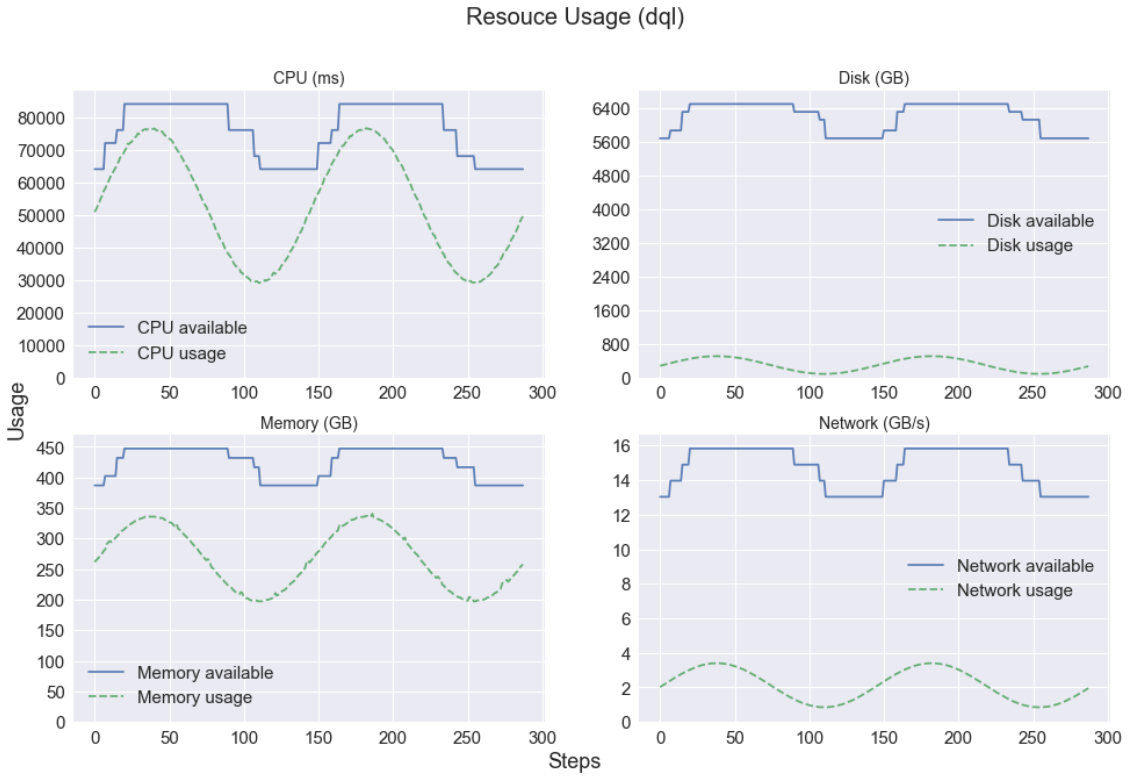


Figure D.1: The DQL algorithm adjusting the resources in a simulation.

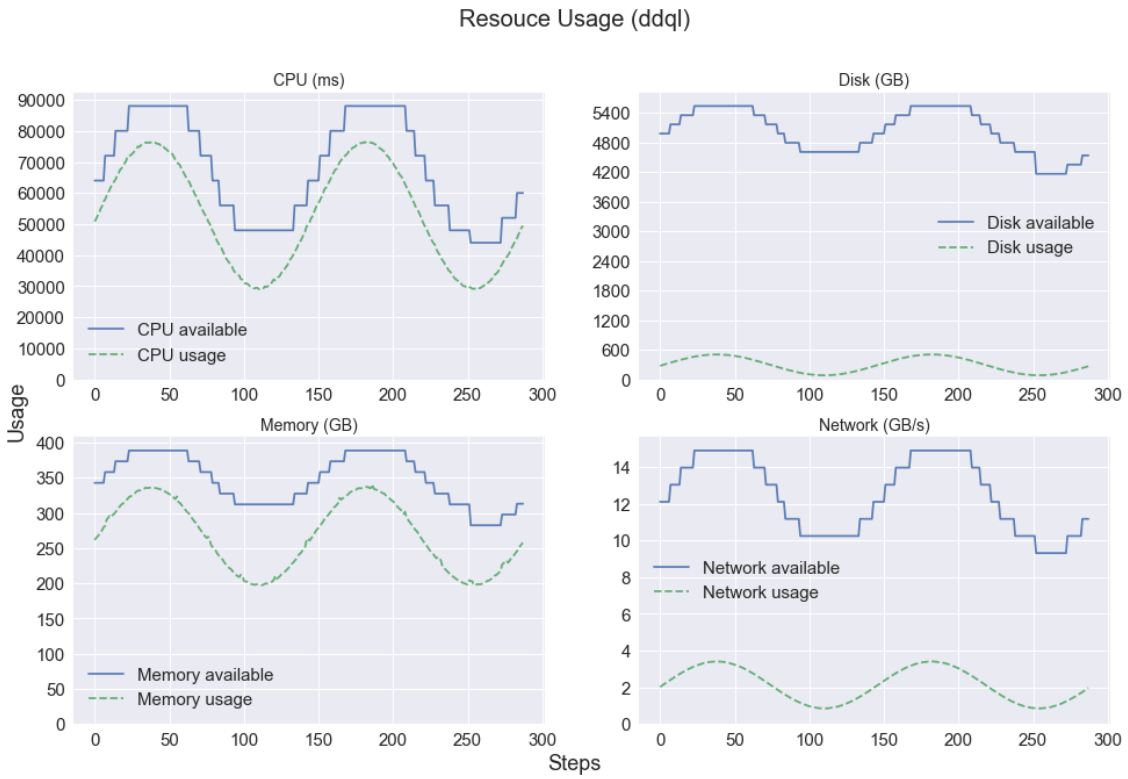


Figure D.2: The DDQL algorithm adjusting the resources in a simulation.

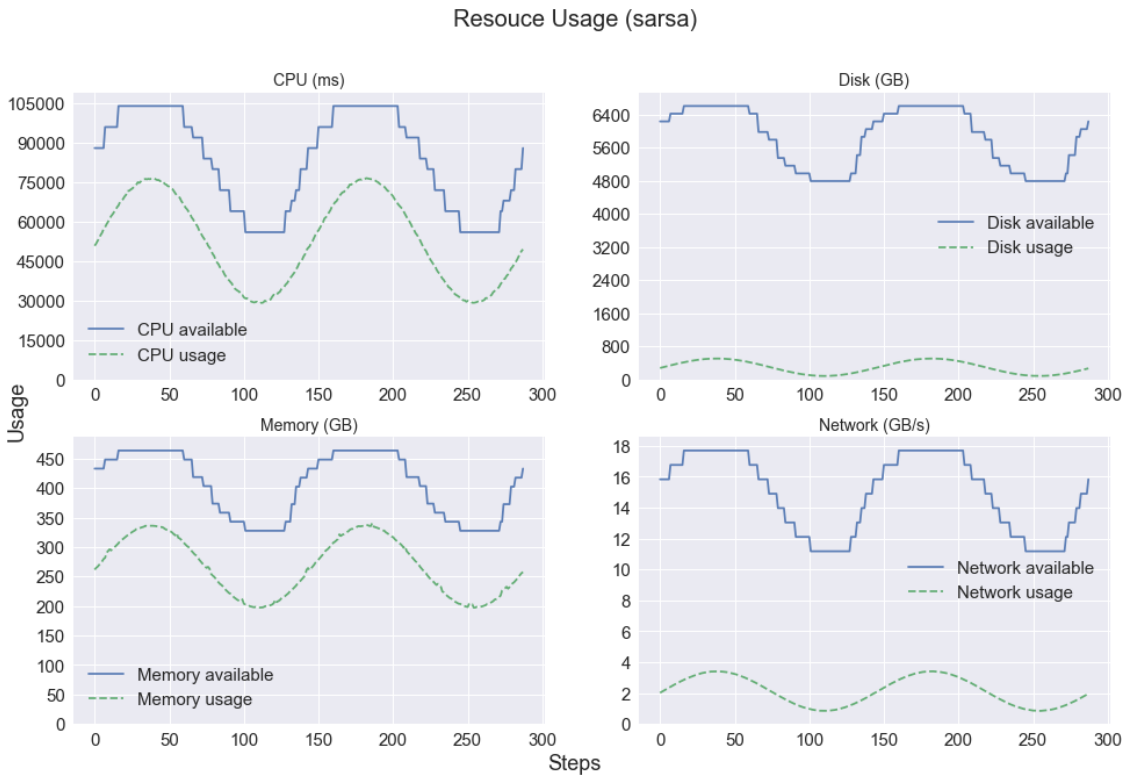


Figure D.3: The SARSA algorithm adjusting the resources in a simulation.

E

Cluster resource usage percentage

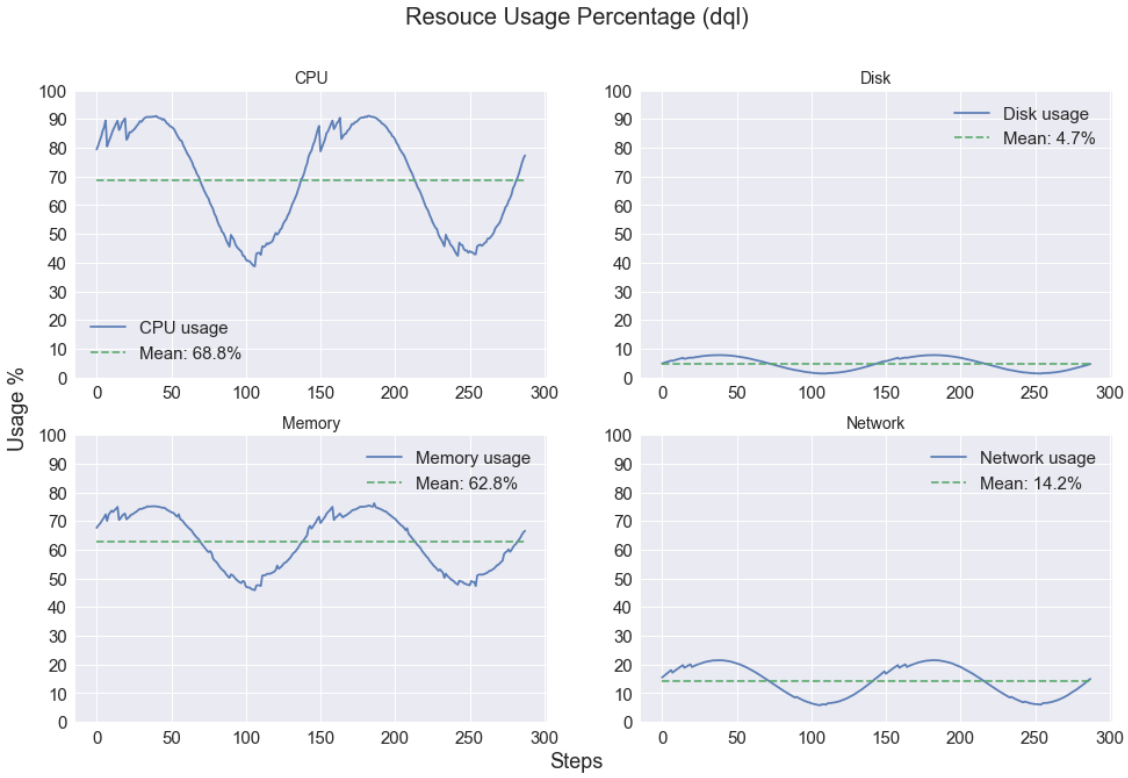


Figure E.1: The DQL algorithm resource usage in percentages each step.

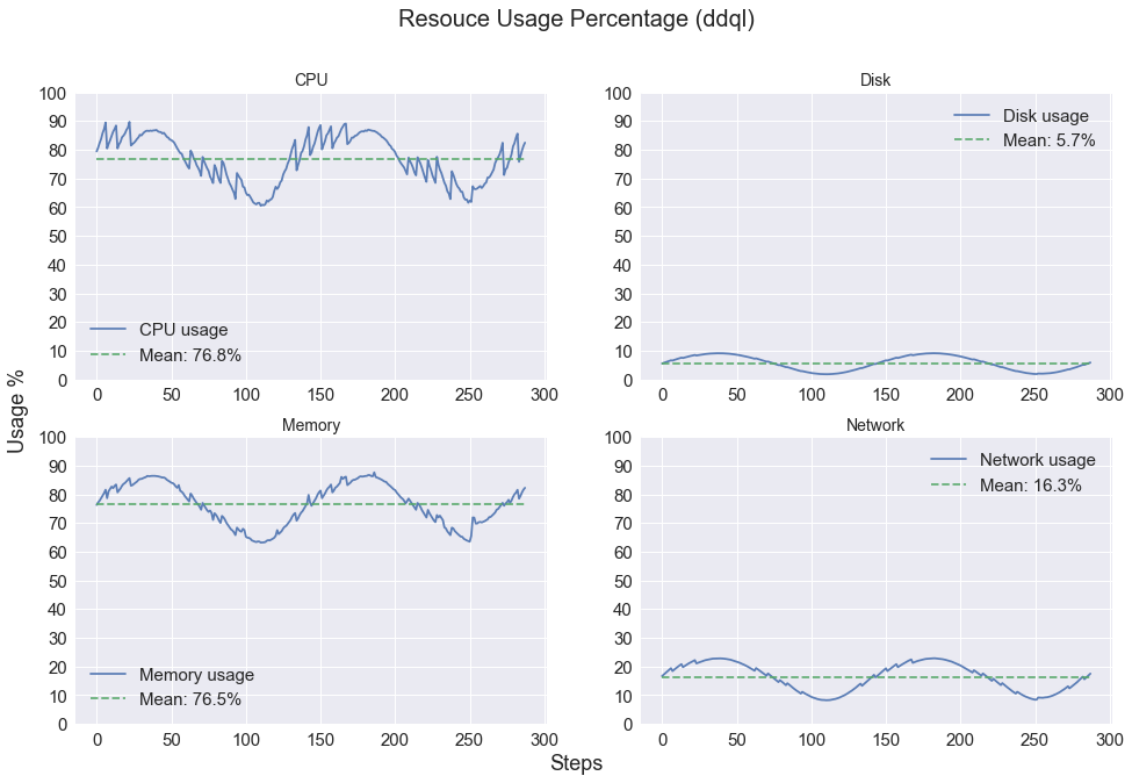


Figure E.2: The DDQL algorithm resource usage in percentages each step.

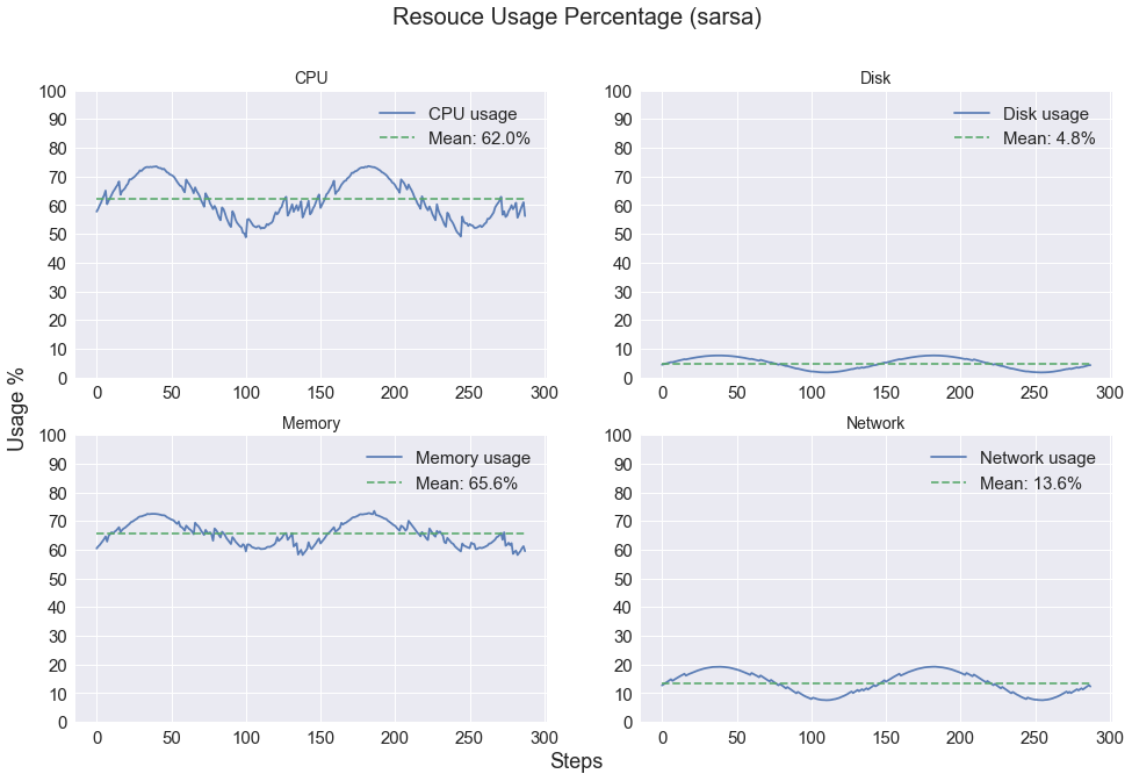


Figure E.3: The SARSA algorithm resource usage in percentages each step.

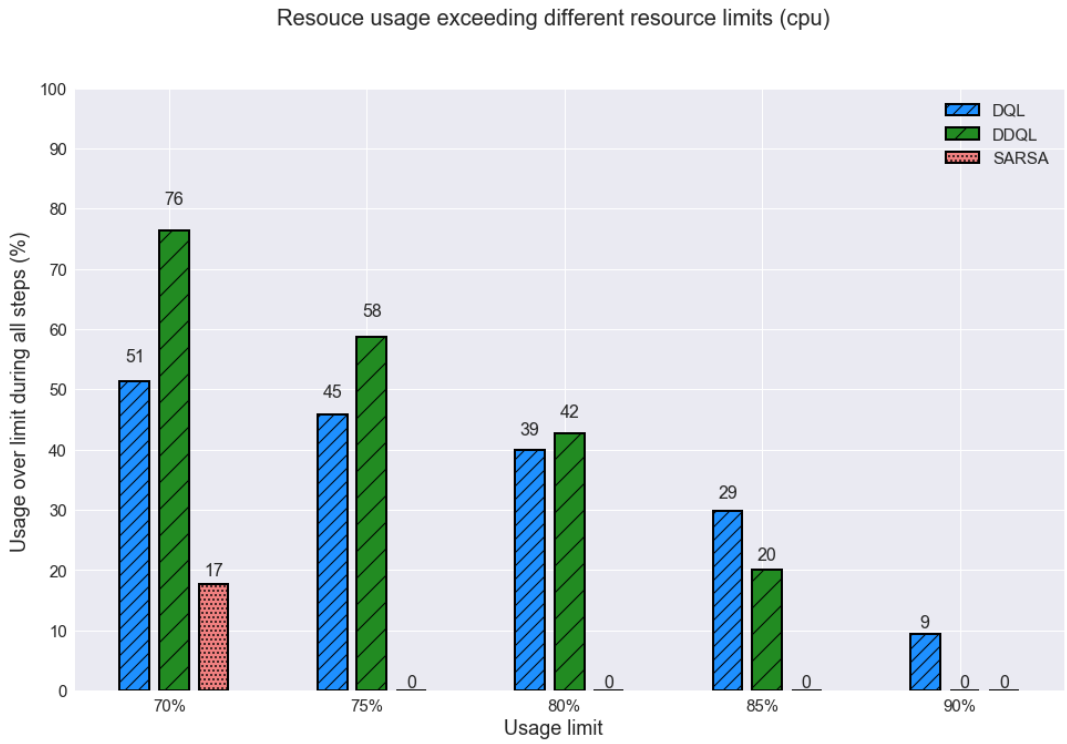


Figure E.4: How much CPU resources exceed a certain limit for each algorithm.

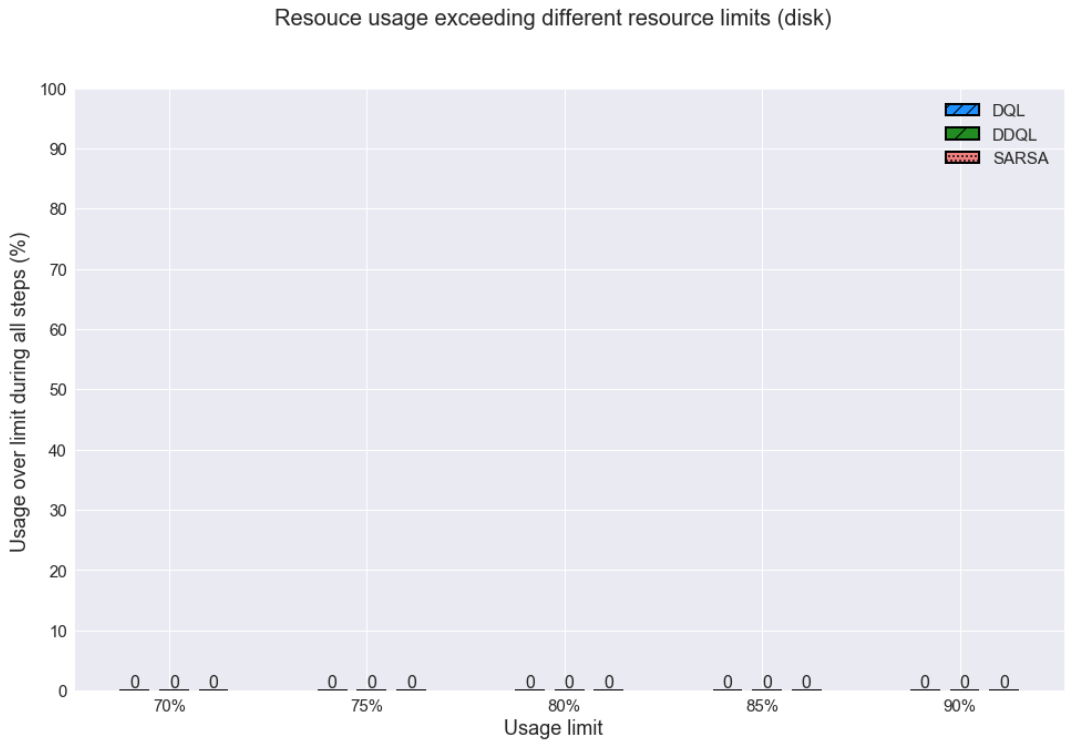


Figure E.5: How much disk resources exceed a certain limit for each algorithm.

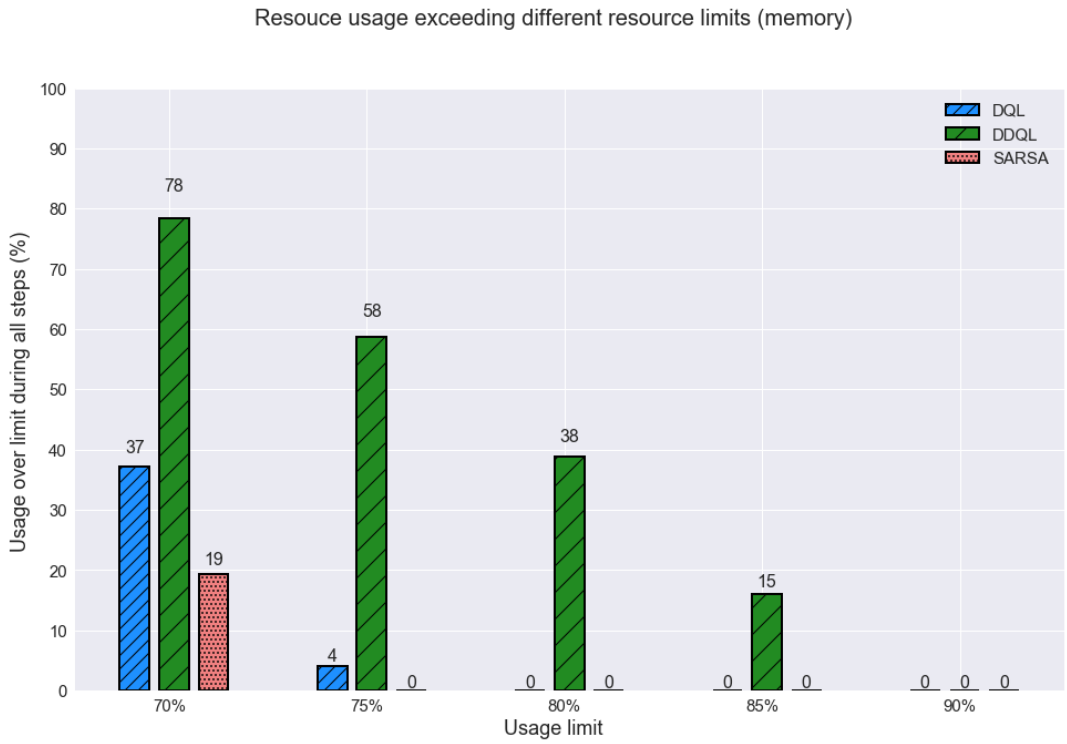


Figure E.6: How much memory resources exceed a certain limit for each algorithm.

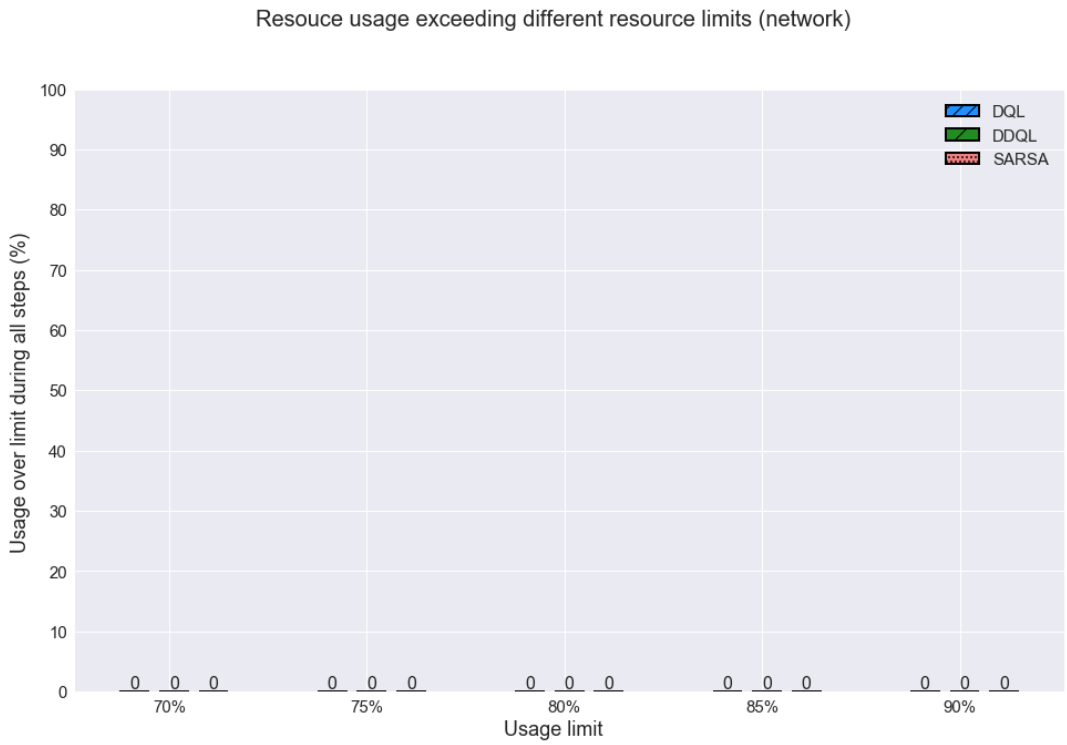
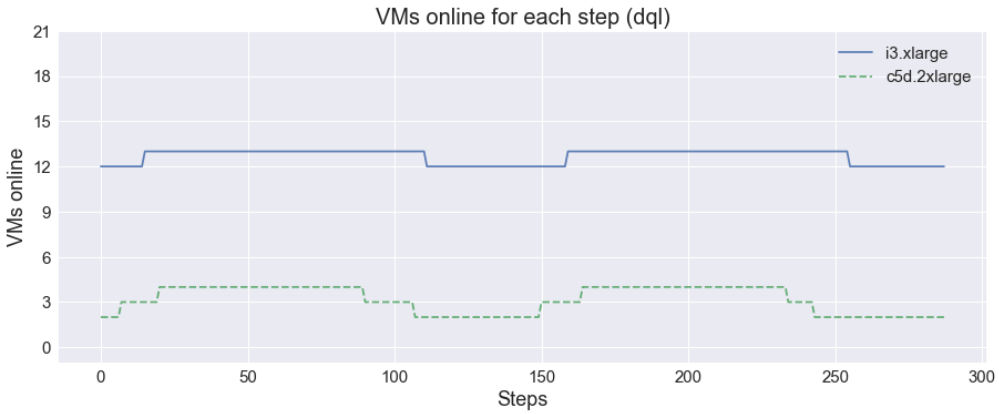
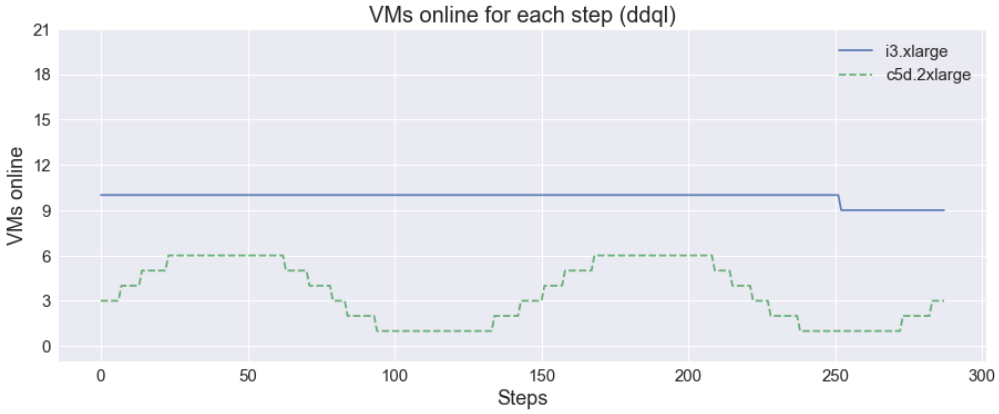


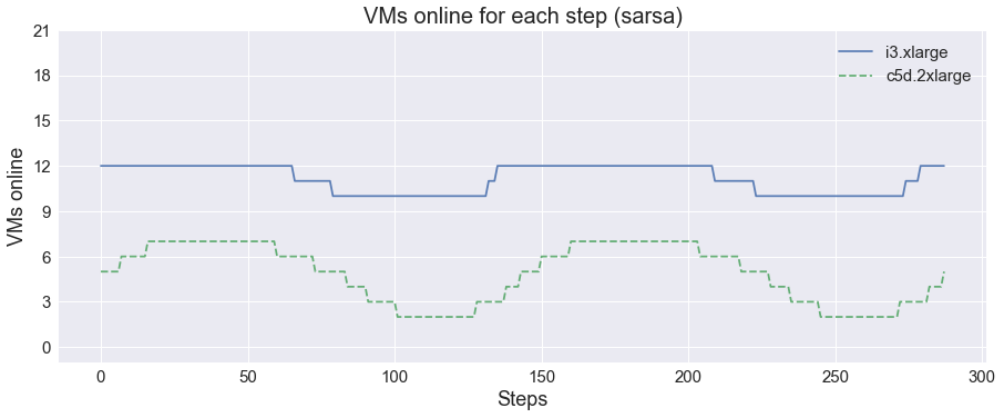
Figure E.7: How much network resources exceed a certain limit for each algorithm.

F

VMs online


VMs online for each step (dql)
i3.xlarge
c5d.2xlarge
VMs online
Steps
0
3
6
9
12
15
18
21
0
50
100
150
200
250
300


VMs online for each step (ddql)
i3.xlarge
c5d.2xlarge
VMs online
Steps
0
3
6
9
12
15
18
21
0
50
100
150
200
250
300


VMs online for each step (sarsa)
i3.xlarge
c5d.2xlarge
VMs online
Steps
0
3
6
9
12
15
18
21
0
50
100
150
200
250
300

G

Cluster Cost