DERP: A Deep Reinforcement Learning Cloud System for Elastic Resource Provisioning

Constantinos Bitsakos, Ioannis Konstantinou and Nectarios Koziris
School of Electrical and Computer Engineering, National Technical University of Athens
Email: {kbitsak, ikons, nkoziris}@cslab.ece.ntua.gr

***Abstract*—Modern large scale computer clusters benefit significantly from elasticity. Elasticity allows a cluster to dynamically allocate computer resources, based on the user's fluctuating workload demands. Many cloud providers use threshold-based approaches, which have been proven to be difficult to configure and optimise, while others use reinforcement learning and decision-tree approaches, which struggle when having to handle large multidimensional cluster states. In this work we use Deep Reinforcement learning techniques to achieve automatic elasticity. We use three different approaches of a Deep Reinforcement learning agent, called DERP (Deep Elastic Resource Provisioning), that takes as input the current multi-dimensional state of a cluster and manages to train and converge to the optimal elasticity behaviour after a finite amount of training steps. The system automatically decides and proceeds on requesting/releasing VM resources from the provider and orchestrating them inside a NoSQL cluster according to user-defined policies/rewards. We compare our agent to state-of-the-art, Reinforcement learning and decision-tree based, approaches in demanding simulation environments and show that it gains rewards up to 1.6 times better on its lifetime. We then test our approach in a real life cluster environment and show that the system resizes clusters in real-time and adapts its performance through a variety of demanding optimisation strategies, input and training loads.**

***Index Terms*—Elasticity, Resource Management, Resource Provisioning, Cloud computing, Deep Reinforecement learning, Double deep Q learning, NoSQL databases, DERP**

I. Introduction

The last years an explosive growth of cloud computing services has taken place. The evolution of cloud technologies and also the large growth of information that needs to be stored and managed created the need of new technologies that could handle these large amounts of data. As a result, traditional SQL databases gave their place to the NoSQL [23] databases and cloud services have emerged. The issue with cloud technologies is the allocation of resources to the cloud users in a way that no resources are being wasted to a user that does not strictly require them at a certain time. In order to achieve that, cloud services are using elasticity [1], a property of cloud computing that allows the services to dynamically allocate resources based on a user's needs.

In this paper we examine elasticity in terms of adding or removing VMs from a user's cluster in order to achieve a desired balance between the cluster's throughput and latency while keeping the costs for the user low. The combination of all of the system parameters (throughput, latency, number of machines, average free memory or disk per machine etc) is defined as the cluster's state in a given point in time. Many approaches have been used to solve the problem of optimal elastic resource provisioning. The issues that occurs with other approaches, as we show in Section II, is that they often fail to generalise over the input and perform efficiently when the number of the input parameters and consequently the size of the space-state increases. Furthermore some of those approaches are difficult to calibrate and optimise.

In this work we present DERP, a system that deals with the elasticity issue by using Deep Reinforcement Learning techniques. DERP is a robust approach that manages to deal with the complex and large size of space-state issue, adapt rapidly to its environment, generalise over the input in a sufficient way, decrease the space consumption needed by former elasticity agents and collect greater rewards in its lifetime when compared to the most efficient of the past approaches presented in Section II.

DERP is based on the Deep Reinforcement learning [2] algorithm introduced by Deepmind [3]. Deepmind used a combination of Deep convolutional neural networks [4] and Q learning algorithm to train an agent that was able to play and achieve sizeable results at a series of Atari games. Since then, lots of updates and improvements were implemented on this idea [5] so now we can talk about agents being able to beat top class DotA players [6], by training themselves in complex screen environments.

We are using three different agents, the *Simple Deep Q learning* agent which implements Deepmind's approach (but with the use of fully connected neural networks [7] instead of convolutional [4] neural networks) as a base approach, upon which we then build the more sophisticated *Full Deep* and *Double Deep Q learning* agents.

II. Related work

The most common way to deal with the issue of elasticity is auto-scaling. Amazon's auto-scaling [8] for instance dynamically increases or decreases a user's resources based on thresholds applied on user cluster's specific metrics. Microsoft's Azure [9] and Celar [10] use the same technique. Yet, as shown in [11], these approaches are difficult to calibrate and optimise.

In [12] the authors use a dynamic programming algorithm that tries to determine through a series of past experiences the optimal behaviour for the system's next-state. Markov Decision Processes (MDPs) [13] and Reinforcement learning [14] algorithms have been used in [15], [16] to address the

978-1-5386-7899-2/18/$31.00 ©2018 IEEE
DOI 10.1109/CloudCom2018.2018.00020

issue, as well as an approach involving wavelets for prediction of a cloud state and resource provisioning. [17]. However, the efficiency of those approaches decreases, as the number of possible states increases. The input parameters of the system (metrics of the cluster) are continuous variables, therefore the number of discrete states can grow exponentially.

To manage this issue in [11] the authors propose an RL approach combined with decision-trees algorithms, in order to split the input parameters based on some split criteria. This approach manages to generalise over the input and to train the agent so that it can find out on its own which state parameters matter to the desired outcome and which not. For instance if the user's policy considers only the high throughput as a desired parameter then the metric of the free memory space per VM does not affect the decisions that should be made by the elasticity agent. Nevertheless, this approach also struggles with large space of states.

III. DERP OVERVIEW

We call **reward** the minimisation of a user's defined policy function that describes the user's preference of the system's behaviour on a mathematical form, as presented in Section IV. We show that our approach collects rewards up to 1.6 times better than the past approaches of the MDPS [15] and the decision-trees [11] agents. The policy function combines the throughput, latency and number of VMs parameters per timestep, in order to fit the user's needs. We show that if the number of our input parameters grows, which means that the number of input states grows, our agents succeed in getting improved and more accurate results. Furthermore, because our input is significantly small in oppose to other deep RL algorithms (most deep RL agents are used to manage image or sound vectors as inputs, meaning thousands of pixels as state parameters), using 13 input parameters, we manage to construct a neural network solution that uses only three hidden layers. Thus our space complexity is significantly smaller than past approaches, which use Q tables [14]. In neural networks all the information is stored on neurons weights and not in other memory or storing units aw we are showing in Section IV.

We are building **DERP**, a cloud based system that handles dynamic resource allocation for a cloud user. The user provides DERP with some policy parameters, meaning the reward function, where he specifies the most crucial parameters for his needs. If a user needs a high throughput for his applications then DERP focuses on maintaining the user's cluster throughput on high levels. If the user needs to keep his cluster at low costs then DERP gives more attention on keeping the number of VMs on the cluster low. DERP then uses its **monitor** module to collect metrics from the cluster every ten seconds (with the use of Ganglia's XML API [18]). The metrics consist of a variety of different parameters such as cluster's throughput, latency, number of VMs, percentage of free memory per VM, percentage of free disc space per VM etc. We use 13 different parameters that describe a state. Then DERP uses its **decision** module, which is either our Simple Deep Q learning, Full Deep Q learning or Double Deep Q learning agent, to determine its next action. We make use of Bellman equation to define our network's Q targets

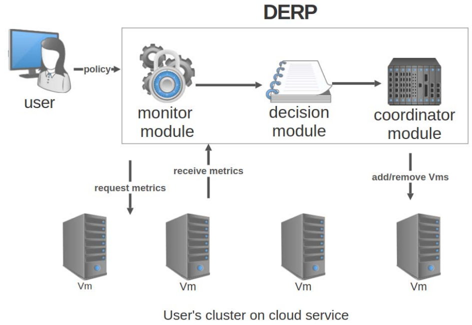


Fig. 1: Derp's Architecture

$$Q*(s,a) = r(s,a) + \gamma maxQ*(s',a'|s,a)$$

where γ is the **discount factor**, that represents the impact of future rewards in our current decision making if we choose a specific action at this timestep. $r(s,a)$ is the reward our agent will get if it takes action a from state s and $Q*(s',a'|s,a)$ is the Q function given that our agent next, is on state s and takes action a. We will elaborate on the use of Bellman equation on Section IV. If an increase or decrease on the cluster's size is decided, then DERP uses its **coordinator** module to add or subtract VMs accordingly.

We can see DERP's architecture in Figure 1.

IV. DERP DECISION ALGORITHMS

The Simple Deep Q learning algorithm that we are using is based on Deepmind's algorithms for Deep Reinforcement learning. Many programs created to efficiently compete with humans in Atari and Go games [19] -as well as in other demanding fields- make use of the aforementioned algorithms. For now we assume a weighted network IV-A, considered as a black box. The replay memory as described in subsection IV-B consists of a buffer that contains N past sequences (s,a,r,s'), that we are going to feed our agent with, on the course of each training lifetime. M is the number of training steps. L is the overall size of the mini batches that we are training our agent with, on every step.

A. Neural Networks

The basic idea behind a neural network is to use the network as a function approximator that computes the output based on a given input. The network's output is then compared to a given target. We want our output to be as close to the target as possible, so we are using Backpropagation [20] to alter our network's weights so that future outputs will be closer to our desired target. Backpropagation deploys gradient descent [21], a first-order iterative optimisation algorithm used to find a function's minimum. The neural network's function is basically the multiplication of an input with the network weights to produce an output.

It is clear that if a certain input parameter does not influence the outcome of a prediction then the weighs that regard this parameters become zero. In the end the weights of the network describe the importance of an input parameter, or the correlations between some input parameters. The above procedure is called training the network. We see that the networks weights hold all the information that a network has absorb during its lifetime.

In the most common example a neural network is used to determine if an image depicts an object of class A or B, so the target is the actual object of the image, that our network's approximator tries to predict its class.

In Deep Reinforcement learning the difference is that there are no specified targets, but we compute our targets using our network to compute the Bellman equation at each state,

$$Q*(s,a) = r(s,a) + \gamma maxQ*(s', a'|s,a)$$

The Q target obtained from Bellman equation, becomes our respective target. So we are using Q learning equations in order to compute some basic targets upon which our network is going to be trained. The $r(s,a)$ factor above is our reward function. We define our reward function as a function that describes the "goodness" of being in a state based on a user's defined parameters.

$$R(s,a) = B*throughput - C*|VMs| - A*lat$$

the user defines the B, C, A parameters. We observe that a cluster's goal is to have sizeable throughput, while obtaining the latency on low levels and minimising the user's cost on resources.

At the early steps of our agent's life, it takes random actions in the environment. We define as **annealing steps**, the number of steps that our agent has to take before reducing the probability of taking a random decision and increasing the probability of taking the optimal decision. This is our e-policy. After $i*annealing-steps$ our algorithm decides to take a random action and not action a, where $a = \arg\max_{a'} Q(s,a')$, with probability $1-i*annealing-steps$. If $i*annealing-steps$ is equal or greater than our training steps, then all of our agent's decisions are optimal (i represents the iteration index). We adopt this strategy to answer the **exploitation vs exploration** dilemma, meaning that if our agent takes the most likely to be the optimal decision from the beginning of its lifetime then it might miss the opportunity to explore the whole environment and obtain better rewards in the future.

We split our data into a training and a testing subset, to evaluate our agent's performance after the training procedure. In this way we make sure that our agent will not overfit on the training data. Overfitting is "the production of an analysis that corresponds too closely or exactly to a particular set of data, and may therefore fail to fit additional data or predict future observations reliably".

B. Experience replay

In order to achieve better results the use of a memory buffer, known as experience replay, has been proposed. This approach suggests that we store some previous memories consisting of a state **s**, an action **a**, the reward **r** that we gain from this action **a** and the resulting state **s'** (s,a,r,s'). In each step of our agent's training life we feed our network with a mini batch of past experienced samples from our memory buffer. This leads to more effective training, since experience is potentially used in a variety of weight updates, allowing for greater data efficiency. Furthermore by receiving random samples from the mini batch, we ensure that the experiences that we feed our network with are not consecutive, thus avoiding strong correlations between the data. Before proceeding to the core algorithm, the first D steps are being utilized for the proper initialization of our replay memory buffer D.

Algorithm 1: Simple Deep Q Learning

```
Initialize replay memory D
Initialize action-value function Q with random values $\theta$
$s = initialstates_1$
**for** $episode = 1$ to $M$ **do**
    Observe state $s$ (by collecting metrics with the monitor module)
    With probability $\epsilon$ select a random action $a_t$ (add/remove VM using coordinator module) otherwise select $a = \arg\max_{a'} Q(s,a')$
    observe reward $r$ and new state $s'$ (by collecting metrics with the monitor module).
    Store sequence $(s,a,r,s')$ at the experience replay buffer
    Sample L number of past experiences $< ss, aa, rr, ss' >$ from our memory buffer and training our agent with them, by calculating the Q targets ($tt$) for each minibatch transitions
        $tt = \begin{cases} rr \quad if \quad ss' \quad is \quad a \quad terminal \quad state \\ rr + \gamma max'_a Q(ss', aa') \\ for \quad non \quad terminal \quad ss' \end{cases}$
    train the Q network using gradient descent with $(tt - Q(ss, aa))^2$ as loss
    $s = s'$
**end for**
```

C. Simple Deep Q learning

This is our basic implementation of Deep Reinforcement learning using experience replay. Our agent's train workflow, presented in algorithm 1, is based on what we already presented in the previous paragraph.

D. Full Deep Q learning

In this approach, presented in algorithm 2, we are using an additional second network. The second network is a clone of our main network, used to compute the target values. It generates the target-Q values that will be used to compute the loss for every action during training. The benefit of using a second network is that at every step of training, the Q-networks values shift, and if we are using a constantly shifting set of values to adjust our network values, then the value estimations can become unmanageable. The network can be destabilised by falling into feedback loops between the target and estimated Q-values. In order to mitigate that risk, the target networks weights are fixed, and periodically updated to the primary Q-networks values. In this way training can proceed in a more

Algorithm 2: Full Deep Q Learning

```
Initialize replay memory D
Initialize action-value function Q with random weights $\theta$
Initialise target function Q' with random weights $\theta$
$s = initialstates_1$
for $episode = 1$ to $M$ do
    Observe state $s$ (by collecting metrics with the monitor module)
    With probability $\epsilon$ select a random action $a_t$ (add/remove VM using coordinator module)
    otherwise select $a = \arg\max_{a'} Q(s, a')$ observe reward $r$ and new state $s'$ (by collecting metrics with the monitor module)
    Store sequence $(s, a, r, s')$ at the experience replay buffer
    Sample L number of past experiences $< ss, aa, rr, ss' >$ from our memory buffer and training our agent with them, by calculating the Q targets ($tt$) for each minibatch transitions
        $tt = \begin{cases} rr \quad if \quad ss' \quad is \quad a \quad terminal \quad state \\ rr + \gamma max'_a Q(ss', aa') \\ for \quad non \quad terminal \quad ss' \end{cases}$
    train the Q network using gradient descent with $(tt - Q(ss, aa))^2$ as loss
    $s = s'$
    Every C-steps reset $Q' = Q$
end for
```

Algorithm 3: Double Deep Q Learning

```
Initialize replay memory D
Initialize action-value function Q with random weights $\theta$
Initialize target function Q' with random weights $\theta$
$s = initialstates_1$
for $episode = 1$ to $M$ do
    Observe state $s$ (by collecting metrics with the monitor module).
    With probability $\epsilon$ select a random action $a_t$ (add/remove VM using coordinator module) otherwise select $a = \arg\max_{a'} Q(s, a')$ observe reward $r$ and new state $s'$ (by collecting metrics with the monitor module).
    Store sequence $(s, a, r, s')$ at the experience replay buffer.
    Sample L number of past experiences $< ss, aa, rr, ss' >$ from our memory buffer and training our agent with them, by calculating the Q targets ($tt$) for each minibatch transitions
        $tt = \begin{cases} rr \quad if \quad ss' \quad is \quad a \quad terminal \quad state \\ rr + \gamma Q(s, argmax(Q(s', a, s''), s')) \\ for \quad non \quad terminal \quad ss' \end{cases}$
    train the Q network using gradient descent with $(tt - Q(ss, aa))^2$ as loss
    $s = s'$
    Every C-steps reset $Q' = Q$
end for
```

stable manner. Every C steps we reevaluate our target networks with our main network values.

E. Double Deep Q learning

The main intuition behind Double Deep Q learning (DDQN) or Double Deep Reinforcement learning (DDRL) is that the regular Deep Q Network often overestimates the Q values of the potential actions to take in a given state. While this would be fine if all actions were always overestimated equally, there is reason to believe this is not the case, as it has been suggested [5]. One can easily imagine that if specific suboptimal actions were regularly assigned higher Q-values than optimal actions, the agent would have a hard time ever learning the ideal policy. In order to correct this, the authors of DDQN [5] proposed a simple solution: instead of taking the maximum over Q-values when computing the target-Q value for our training step, we use our primary network to choose an action, and our target network to generate the target Q-value for that action. By decoupling the action choice from the target Q-value generation, we are able to substantially reduce the overestimation, and train faster and more reliably. We adopt and customise this approach to create our Double Deep Q learning agent, presented in algorithm 3. The new DDQN equation that we use for updating the target value is:

$$Q_{target} = r + Q(s, argmax(Q(s', a, s''), s'))$$

V. Network Architecture

A. Architecture

We are using Google's Tensorflow framework [22] to build and train our neural network agent. Our network consists of three hidden layers (example in Figure 2). This means that the computer space that our agent consumes is reduced adequately. The first layer consist of 64 neurons, the second of 128 and the third of 256. We use a fully connected and not convolutional NN. The second one is commonly used upon deep RL applications, however since we do not have image vectors as input, the fully connected layers approach works as sufficiently, as our input data cannot be considered spatial. Convolutional neural networks manage to handle spatial data efficiently and discover hidden correlations between them [23]. In our **Simple Deep Q learning** agent we use only this network. When constructing **Full Deep Q learning** and **Double Deep Q learning** agents, we use two identical networks one to compute our values and one to compute our targets, that only differ on their weights values, as we showed in Section IV.

We are using an **experience replay buffer** which can store up to 500 different tuples consisting of a state **s**, an action **a**, the reward **r** that we collect for this action **a** and the state **s'** that our agent finds itself in after taking the action **a** (s,a,r,s'). At the first 360 steps of our algorithm which we call **pretrain steps** we are just filling the buffer with memories. After that we are feeding our agent with as many past experiences as our **batch size** parameter suggests.

As a trainer we use the Tensorflow RMSPropOptimizer [24]. RMSProp can be seen as a generalisation of Rprop [25] and is capable to work with mini batches as well opposed to only full batches.

VI. Simulations

We are running two different demanding simulations, for testing purposes.

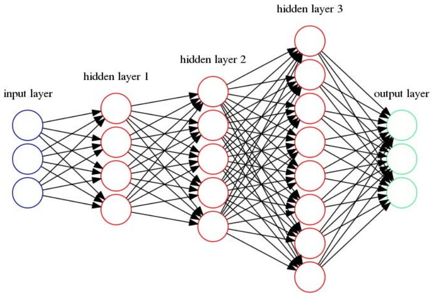


Fig. 2: Example of a three hidden layer neural network

A. Simple simulation scenario

Firstly we are using a simulation scenario from the field of cloud computing, which we call the “simple scenario” for convenience. In this scenario, the agent is asked to make elasticity decisions that resize a cluster running a database under a varying incoming load. The load consists of read and write requests, and the capacity of the cluster depends on its size as well as the percentage of the incoming requests that are reads. Specifically:

- The cluster size can vary between 1 and 20 virtual machines
- The available actions to the agent in each step are to increase the size of the cluster by one, decrease the size of the cluster by one, or do nothing.
- The incoming load is a sinusoidal function of time: $load(t) : 50 + 50sin(\frac{2\pi}{250})$
- The percentage of incoming requests that are reads is a sinusoidal function of time with a different period: $r(t)$: $0.75 + 0.25sin(\frac{2\pi}{340})$
- If $vms(t)$ is the number of virtual machines currently in the cluster, the capacity of the cluster is given by: $capacity(t) = 10vms(t)r(t)$
- The reward for each action depends on the state of the cluster after executing the action and is given by: $Rt = min(capacity(t+1), load(t+1)) - 3vms(t+1)$.
- Training steps ∈ [2000, 5000, 10000, 20000, 500000]
- Evaluation steps=2000
- max error: 10^6 (the largest acceptable value of $loss = (target - prediction)^2$)

As we can see the reward function encourages the agent to increase the size of the cluster to the point where it can fully serve the incoming load, but punishes it for going further than that. In order for the agent to behave optimally, it needs to not only identify the way its actions affect the clusters capacity and the dependence on the level of the incoming load, but also the dependence on the types of the incoming requests. An important goal of our algorithm is to recognise which input parameters matter for the outcome and which are not and also discover correlations between the input parameters. So we feed our agent with 7 more randomly valued parameters (meaning that our agent should be able to find out on its own that these parameters do not influence the outcome). Before performing our experiments we calibrate our network after testing it to the simulation environment with 5000 training steps. We conclude that the best parameterization given the simulations is to pick a value of about 360 experience batch size for our memory buffer, a value of about 0.00025 for our backpropagation’s learning rate [20] and a value of annealing steps equal to our training steps/10 every time, so that our e value decreases linearly to our input.

B. Complex simulation scenario

After observing that our agents perform efficiently in the “simple simulatio”, as we show in sec VI-C, we test them in a more demanding environment, which we call the “complex simulation scenario”. Cluster setup:

- The cluster size can vary between 1 and 20 virtual machines
- The available actions to the agent in each step are to increase the size of the cluster by one, decrease the size of the cluster by one, or do nothing.
- The incoming load is a sinusoidal function of time: $load(t) : 50 + 50sin(\frac{2\pi}{250})$
- The percentage of incoming requests that are reads is a sinusoidal function of time with a different period: $r(t)$: $0.75 + 0.3sin(\frac{2\pi}{340})$
- I/O operations per second: $IO(t) : 0 : 6 + 0 : 4sin(\frac{2\pi}{195})$
- I/O penalty:

$$IOpen(t) = \begin{cases} 0 \quad if \quad 0.7 > IO(t) \\ IO(t) - 0.7 \quad if \quad 0.7 < IO(t) < 0.9 \\ 0.2 \quad if \quad IO(t) > 0.9 \end{cases} \tag{1}$$

- If vms(t) is the number of virtual machines currently in the cluster, the capacity of the cluster is given by: $capacity(t) = 10vms(t)r(t)$
- The reward for each action depends on the state of the cluster after executing the action and is given by: $Rt = min(capacity(t+1), load(t+1)) - 3vms(t+1)$.
- Training steps ∈ [2000, 5000, 10000, 20000, 500000]
- Evaluation steps=2000
- Learning rate: 0.00025
- batch size: 360
- max error: 10^6 (the largest acceptable value of $loss = (target - prediction)^2$)

To increase the difficulty of this scenario, we have increased the effect of the types of the queries to the capacity of the cluster, and have also added one more parameter that affects the behaviour of the system in a non-linear manner, namely the I/O operations per second. This parameter takes values between 0.2 and 1.0, but only affects the performance of the cluster if its value is higher that 0.7 by adding a penalty to the performance of each VM. Just like in the simple scenario we included 6 additional random input parameters. Three of them followed a uniform distribution within [0, 1], and another three took integer values within [0, 9] with equal probability.

C. Simulation results

We use the network that we have now fully described to test our three agents. We compare our agents to MDP and

Q-learning approaches as presented in [15] and to MDDPT and QDT approaches as presented here [11]. The last two approaches are an MDP and a Q learning algorithm combined with decision trees for better clustering over the input. It is worth mentioning that these last two algorithms were the most efficient approaches on the elasticity issue to the point that we were building DERP as suggested here [11]. We test all algorithms for a variety of different size of training steps.

1) Simple simulation scenario: Firstly we test our agents in the "simple simulation scenario" as presented above. In Figure 3a we show the performance of our Double Deep RL when trained for 5000 steps. We show the results of the past approaches and our approaches at Figures 3b and 3c. And we show the comparison between our best agent Double Deep Rl and the best past approaches in Figures 3d and 3e.

2) Complex simulation scenario: Now we are testing our agents in the "complex simulation scenario" as presented above. In Figure 4a we show the performance of our Double Deep RL agent when trained for 5000 steps. We show the results of the past approaches, our approaches at Figures 4b and 4c. And we show the comparison between our best agent Double Deep RL and the best past approaches in Figures 4d and 4e.

3) Meaning of the diagrams: At the diagrams that we present the effectiveness of our agent (Figures 3a and 4a) we present with a red line our cluster's incoming load and with blue dots our cluster's number of VMs. We also present our reward for each experiment. Most of the times the reward function is directly proportional to the size of the incoming load and inversely proportional to the number of our cluster's VMs. This is why we choose this representation. At the diagrams that we show comparisons between agents (Figures 3e, 3d, 3b, 3c , 4e, 4d, 4b and 4c), we show the total gain of rewards of different agents in their lifetime, with different amount of training steps at each time (x-axis).

D. Conclusions of simulation results

Our Deep RL agents carry a lot of advantages as opposed to former algorithms used for elasticity as we showed experimentally for some [15], [11] and we discussed theoretically in Section I for others. The most important being:

1) Adaptation: The adaptation of Deep RL model is robust and rapid, regardless the size of the training set. As we notice in former algorithmic models used, the more complex models [11] needed larger set of data before fully adapting to the problem and finding the optimal solution, whereas more simple algorithmic approaches [15], adapted more quickly but did not manage to find the optimal decision the most times in order to obtain a more sufficient reward.

2) Clustering over the input: DERP's agents manage to cluster the data and find which input data truly matters to the outcome of the algorithm and which not. Deep RL algorithms can do that at any set of data, regardless of its size. They do not need any number of states provided and as much more data we can provide our agents (including the number of input parameters) the better for its effectiveness. As is shown here [2], deep learning agents deal with complex screen image vectors, meaning that each state is described by a significant variety of different pixels (parameters).

3) Space consumption: The space complexity of the DERP's approach gives it a large edge against other approaches. Thus because all other RL approaches need computer space for storing every past experience or subsets of past experiences. The attempt of this [11] decision-trees approach was to find a way to manage the data without using to much computer space and at the other hand extract as much information as possible. Deep Rl uses Neural Networks which give an important advantage in this aspect, as neural networks do not use additional computer space for storing every past experience as all the information needed is stored in the networks weights. And as we previously showed, our network consist of only three hidden-layers, minimising storage space. That gives us the elasticity to handle as much data, experience and information as we possible. In our approach we do use the experience replay feature as shown in Section IV but the space it needs is proportionally limited and with a upper bound.

4) Total rewards: Aas we can see in our examples our Full DQN agent and our Double DQN agent manage to get sufficient rewards, even better from all the previous approaches that we compared our agents with. DERP surpasses its predecessors in terms of sufficiency, efficiency, adaptation, storage economy and scaling.

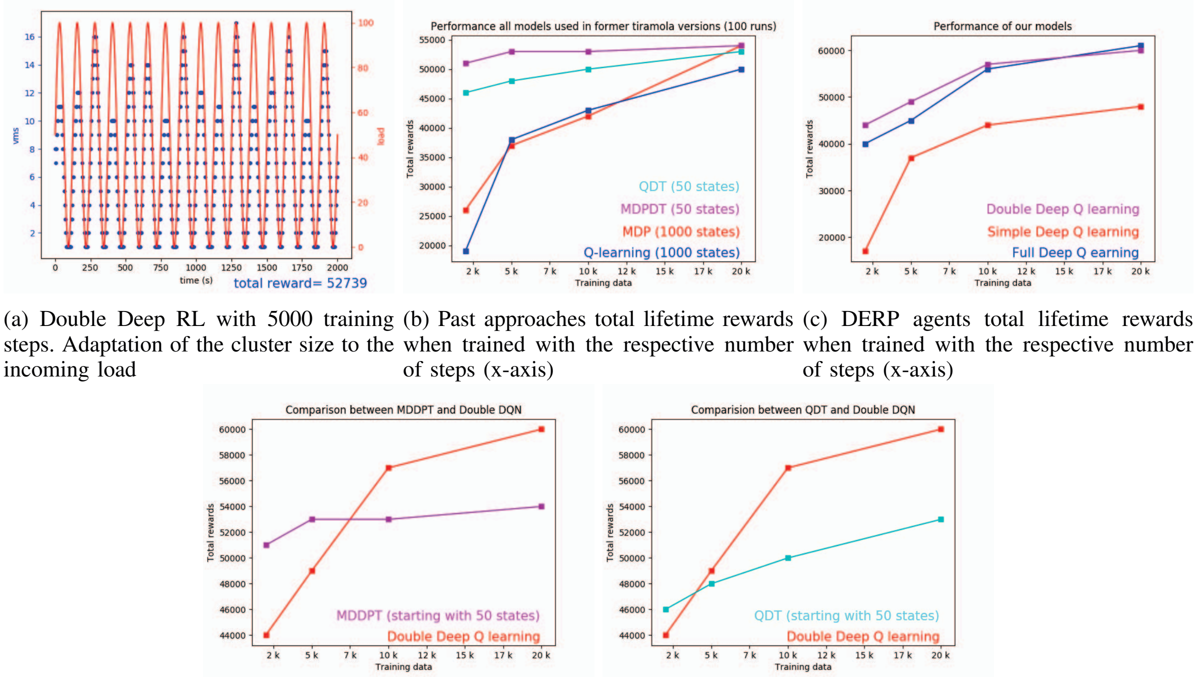
VII. Experimental results

In this Section we are going to present the experiments we performed with DERP in real life environments. We are using Okeanos service [26] as our cloud infrastructure. We have built a Cassandra cluster upon Okeanos containing 16 VMs, which we trigger with different workloads produced by YCSB service [27].

Our workloads are sinusoidal reads and writes on our Cassandra database. The load of those reads and writes follows a sinusoidal distribution. The percentage of reads or writes requests is random. We use every single node of our cluster as a receiver of our requests as in Cassandra every node can serve requests and there is no central node. Every ten seconds we collect metrics with ganglia XML API [18] and use our DERP agent to determine the best decision on every step of the procedure, after we have trained it. The programming environment that we use is Anaconda [28] and the library that provides us the neural network and training tools is Google's Tensorflow [22].

The metrics that we take represent the cluster current state and consist of the following parameters:

- The number of active VMs on the cluster.
- The latency of the cluster.
- The throughput of the cluster.
- The amount of cached memory on the cluster.
- The current number of operations-requests served by the cluster at the current point.
- The number of operations-requests served by the cluster on the last state. We use this information to determine if



(a) Double Deep RL with 5000 training steps. Adaptation of the cluster size to the incoming load

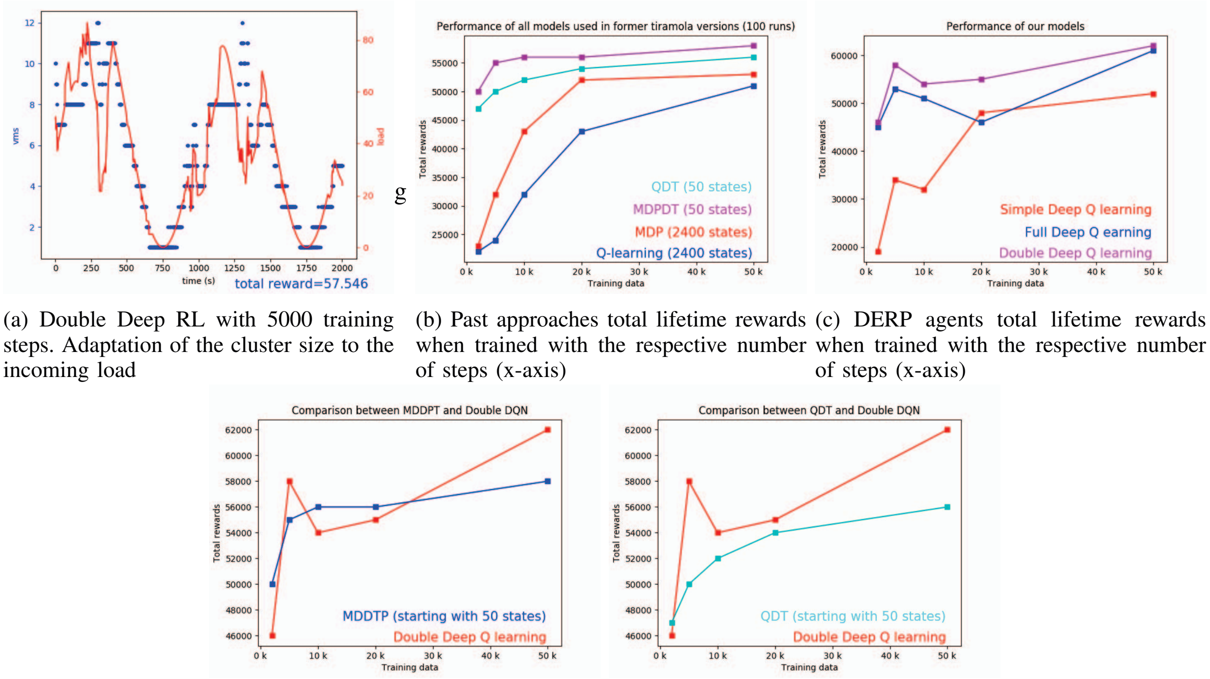
(b) Past approaches total lifetime rewards when trained with the respective number of steps (x-axis)

(c) DERP agents total lifetime rewards when trained with the respective number of steps (x-axis)

(d) MDPDT algorithm [11] vs DERP's DDQN agent in the simple scenario

(e) QDT algorithm [11] vs DERP's DDQN agent in the simple scenario

Fig. 3: Various results on a simple simulation scenario



(a) Double Deep RL with 5000 training steps. Adaptation of the cluster size to the incoming load

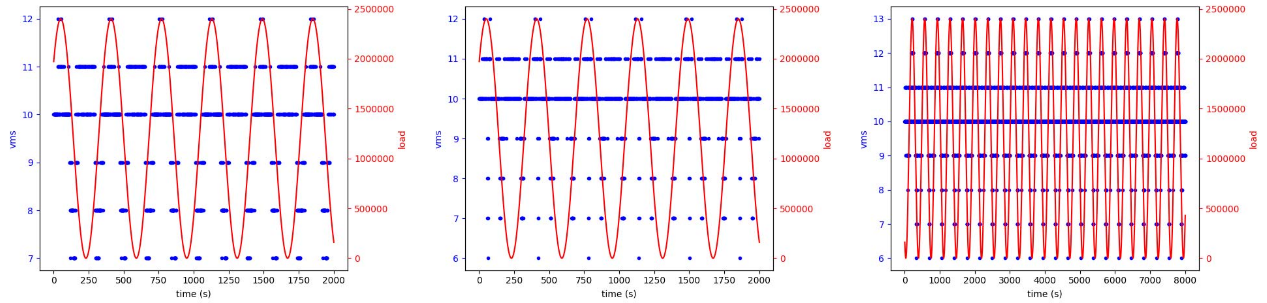
(b) Past approaches total lifetime rewards when trained with the respective number of steps (x-axis)

(c) DERP agents total lifetime rewards when trained with the respective number of steps (x-axis)

(d) MDPDT algorithm [11] vs DERP's DDQN agent in the complex scenario

(e) QDT algorithm [11] vs DERP's DDQN agent in the complex scenario

Fig. 4: Various results on a complex simulation scenario



(a) Simple Deep RL with 20000 training steps. Adaptation of the cluster size to the incoming load

(b) Full Deep RL with 20000 training steps. Adaptation of the cluster size to the incoming load

(c) Double Deep RL with 60000 training steps. Adaptation of the cluster size to the incoming load

Fig. 5: Experimental results of DERP's agents

the size of operations is currently more possible to be on an increasing or decreasing slope.

- The amount of free memory on the cluster.
- The percentage of cluster CPU idle.
- The cluster amount of buffered memory.
- The cluster amount of available memory.
- The cluster amount of shared memory.
- The cluster amount of free disc space.
- The amount of bytes in, that each node of the cluster experience in the current state.
- The amount of bytes out, that each node of the cluster experience in the current state.

As we can see we use many input parameters to describe our state as to demonstrate the ability of DERP to determine which of those input parameters matter to its future decisions and which are not. Our reward function is

$$Reward = 0.01 * throughput - 0.00001 * latency - 2 * VMs$$

Our network is described as follows:

- Training steps: 20000
- Evaluations steps: 2000
- Annealing steps: 2000
- Pretrain steps: 620
- max error: 10^6
- Batch size: 360 experiences
- Our learning rate: 0.00025

A. Evaluation

After the training part, we test our DERP's agents for 2000 execution steps. We test it for our three different approaches, Simple DQN, Full DQN and Double DQN. We observe that our agent rapidly converges its behaviour to the optimal, obtaining sizeable rewards. When Simple DQN is the case, our agent, spends more time until it finds the optimal solution. We see the results in Figures 5a and 5b. Then we test our best agent, Double DQN with a biggest dataset for training, containing 60000 different states. We see the result in Figure 5c.

VIII. Conclusions

In this work we presented a Deep Reinforcement Learning agent for cloud elasticity problems, called DERP, by combining cutting edge algorithmic techniques in both deep learning and cloud resource management areas. Our results indicate that our agent outperforms the previous state-of-the-art approaches that we compared it with in our work, [15] [11], in terms of clustering the input data, managing large number of input states, as well as collecting sufficient rewards on its lifetime. To summarise, the advantages of our approach are the following:

- DERP can learn and perform tasks in large environments where each state depends on multiple continuous parameters. In addition, DERP does not demand space partitioning or data clustering and does not experiences issues when dealing with large input datasets. This gives DERP an advantage to past approaches that we compared it with, which struggled with large datasets.
  DERP manages to determine by its own which input parameters are significant for its future decisions as shown in the simulations and our conclusions in Section VI.
- DERP shows that we can efficiently use Deep Reinforcement learning outside the field of image-related problems where it is usually used, and achieve adequate results even in an environment where the state is constructed by a small number of parameters. Our agent behaved optimally in a cloud environment where it provided each user with the best resource provisioning based on his predefined needs.
- DERP adapts rapidly to its environment and manages to converge to the optimal behaviour.
- DERP manages to collect rewards at average 1.6 times better than past approaches that we compared it with, at its lifetimes experiences in different environments simple or complex, simulated or real life.

In conclusion, DERP is the first cloud-based Deep Reinforcement Learning agent that manages to adequately adjust to a cloud user behaviour, predict his next moves and needs and follow an optimal resource management policy. It deals optimally with the large, continuous datasets, an issue that past approaches struggled with. It manages to rapidly determine which input parameters are important for its future decisions, it uses considerably less computer space than past approaches and it manages to collect 60 % better rewards at an average case study compared to the past approaches. Furthermore it can easily scale to large datasets (without consuming substantially more computer space) and collect sufficient rewards.

Acknowledgement

This paper is supported by European Union's Horizon 2020 Framework Programme - Grant Agreement Number: 780245, project E2Data (European Extreme Performing Big Data Stacks).

References

[1] P. Mell and T. Grance, "The NIST Definition of Cloud Computing," *National Institute of Standards and Technology*, vol. 53, no. 6, p. 50, 2009.

[2] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. A. Riedmiller, "Playing Atari with Deep Reinforcement Learning," *CoRR*, vol. abs/1312.5602, 2013.

[3] "Deepmind," https://deepmind.com/.

[4] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems 25*, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, Eds. Curran Associates, Inc., 2012, pp. 1097–1105. [Online]. Available: http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf

[5] H. v. Hasselt, A. Guez, and D. Silver, "Deep Reinforcement Learning with Double Q-Learning," in *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, ser. AAAI'16, 2016, pp. 2094–2100.

[6] "AI Beats the World's Best Go Gamers after just two Weeks of Learning," https://newatlas.com/open-ai-dota2-machine-learning/50882/.

[7] "Fully Connected Neural Networks," https://towardsdatascience.com/under-the-hood-of-neural-networks-part-1-fully-connected-5223b7f78528.

[8] "AWS, Amazon Autoscaling," https://docs.aws.amazon.com/autoscaling/ec2/userguide/as-scale-based-on-demand.html.

[9] "Microsoft's Azure," https://docs.microsoft.com/en-us/azure/monitoring-and-diagnostics/insights-autoscale-common-metrics.

[10] I. Giannakopoulos, N. Papailiou, C. Mantas, I. Konstantinou, D. Tsoumakos, and N. Koziris, "Celar: Automated application elasticity platform," in *2014 IEEE International Conference on Big Data (Big Data)*, Oct 2014, pp. 23–25.

[11] K. Lolos, I. Konstantinou, V. Kantere, and N. Koziris, "Elastic management of cloud applications using adaptive reinforcement learning," in *Big Data (Big Data), 2017 IEEE International Conference on*. IEEE, 2017, pp. 203–212.

[12] R. Taft, N. El-Sayed, M. Serafini, Y. Lu, A. Aboulnaga, M. Stonebraker, R. Mayerhofer, and F. Andrade, "P-Store: An Elastic Database System with Predictive Provisioning," in *Proceedings of the 2018 International Conference on Management of Data*, ser. SIGMOD '18, New York, NY, USA, 2018, pp. 205–219.

[13] M. L. Puterman, *Markov Decision Processes: Discrete Stochastic Dynamic Programming*, 1st ed. New York, NY, USA: John Wiley & Sons, Inc., 1994.

[14] C. J. C. H. Watkins and P. Dayan, "Q-Learning," *Machine Learning*, vol. 8, no. 3, pp. 279–292, 1992.

[15] D. Tsoumakos, I. Konstantinou, C. Boumpouka, S. Sioutas, and N. Koziris, "Automated, Elastic Resource Provisioning for NoSQL Clusters Using Tiramola," in *Cluster, Cloud and Grid Computing (CCGrid), 2013 13th IEEE/ACM International Symposium on*. IEEE, 2013, pp. 34–41.

[16] J. Rao, X. Bu, C.-Z. Xu, L. Wang, and G. Yin, "VCONF: A Reinforcement Learning Approach to Virtual Machines Auto-Configuration," in *Proceedings of the 6th International Conference on Autonomic Computing*, ser. ICAC '09, New York, NY, USA, 2009, pp. 137–146.

[17] H. Nguyen, Z. Shen, X. Gu, S. Subbiah, and J. Wilkes, "AGILE: Elastic Distributed Resource Scaling for Infrastructure-as-a-Service," in *Proceedings of the 10th International Conference on Autonomic Computing (ICAC 13)*, San Jose, CA, 2013, pp. 69–82.

[18] M. L. Massie, B. N. Chun, and D. E. Culler, "The Ganglia Distributed Monitoring System: Design, Implementation, and Experience," *Parallel Computing*, vol. 30, no. 7, pp. 817–840, 2004.

[19] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis, "Mastering the Game of Go with Deep Neural Networks and Tree Search," *Nature*, vol. 529, pp. 484–503, 2016.

[20] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning Internal Representations by Error Propagation," California Univ San Diego La Jolla Inst for Cognitive Science, Tech. Rep., 1985.

[21] L. Bottou, "Gradient descent." [Online]. Available: http://leon.bottou.org/publications/pdf/compstat-2010.pdf

[22] "Tensorflow," https://www.tensorflow.org/.

[23] Z. Zuo, B. Shuai, G. Wang, X. Liu, X. Wang, B. Wang, and Y. Chen, "Convolutional Recurrent Neural Networks: Learning Spatial Dependencies for Image Representation," in *2015 IEEE Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, June 2015, pp. 18–26.

[24] "Rmspropoptimizer," https://www.tensorflow.org/api_docs/python/tf/train/RMSPropOptimizer.

[25] M. Riedmiller and H. Braun, "A Direct Adaptive Method for Faster Backpropagation Learning: The RPROP Algorithm," in *IEEE INTERNATIONAL CONFERENCE ON NEURAL NETWORKS*, 1993, pp. 586–591.

[26] "Okeanos Cloud service," https://okeanos.grnet.gr/home/.

[27] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking Cloud Serving Systems with YCSB," in *Proceedings of the 1st ACM symposium on Cloud computing*, 2010, pp. 143–154.

[28] "Anaconda Environment," https://anaconda.org/anaconda/python.