Spring 5-22-2017

Generic Online Learning for Partial Visible & Dynamic Environment with Delayed Feedback

Behrooz Shahriari
San Jose State University

Generic Online Learning for Partial Visible & Dynamic Environment with Delayed Feedback

Online Learning for 5G Network Load-Balancer

A Writing Project

Presented to

The Faculty of the Department of Computer Science

San Jose State University

In Partial Fulfillment of the Requirements for the Degree

Master of Computer Science

By

Behrooz Shahriari

Spring 2017

The Designated Committee Approves the Writing Project Titled

Generic Online Learning for Partial Visible & Dynamic Environment with Delayed Feedback

By Behrooz Shahriari

APPROVED FOR THE DEPARTMENT OF COMPUTER SCIENCE SAN JOSÉ

STATE UNIVERSITY

April 2017

Dr. Melody Moh Department of Computer Science

Dr. Teng Moh Department of Computer Science

Dr. Sami Khuri Department of Computer Science

Abstract

Reinforcement learning (RL) has been applied to robotics and many other domains which a system must learn in real-time and interact with a dynamic environment. In most studies the state-action space that is the key part of RL is predefined. Integration of RL with deep learning method has however taken a tremendous leap forward to solve novel challenging problems such as mastering a board game of Go. The surrounding environment to the agent may not be fully visible, the environment can change over time, and the feedbacks that agent receives for its actions can have a fluctuating delay. In this paper, we propose a Generic Online Learning (GOL) system for such environments. GOL is based on RL with a hierarchical structure to form abstract features in time and adapt to the optimal solutions. The proposed method has been applied to load balancing in 5G cloud random access networks. Simulation results show that GOL successfully achieves the system objectives of reducing cache-misses and communication load, while incurring only limited system overhead in terms of number of high-level patterns needed. We believe that the proposed GOL architecture is significant for future online learning of dynamic, partially visible environments, and would be very useful for many autonomous control systems.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACRYNOMS AND ABBREVIATIONS

IS	Intelligent Systems
RL	Reinforcement Learning
MDP	Markov Decision Process
SARSA	State-Action Reward State-Action
TD	Temporal Difference
FIS	Fuzzy Inference System
DFQL	Dynamic Fuzzy Q-Learning
DL	Deep Learning
ALE	Arcade Learning Environment
AGI	Artificial General Intelligence
GA	Genetic Algorithm
GOL	Generic Online Learning
IC	Input Channel
OC	Output Channel
EN	Numerical Entity
GN	Generic Entity
EM	Medium Entity
HE	High Entity
HC	High-level Channel
eN	entity-Node
LB	Load Balancer
VM	Virtual Machine
CS	Cloud Storage
ST	Synchronization Time
MB	Memorized-BestVM
MB-F	Memorized-BestVM with Fixed ST

Introduction

Intelligent systems (IS) have the capability to observe their surrounding environment through their sensory input channels and interact with the environment via their output channels, which their actions directly or indirectly affect the environment. Also, a realistic environment is dynamic: with time it changes itself, as well as the elements it depends on.

The intelligent system is built to achieve a set of goals. IS in each time-frame requires to make a decision based on the observed data of its surrounding environment. An IS system requires to adapt in real-time based on the feedback it received. Feedback can be interpreted as some changes in observed data from an environment based on actions of the system; however, the feedback of a particular action may reach to the system with some delay, or the system may have no knowledge of mapping between observed feedback and its previous actions.

Reinforcement learning (RL), is one of the best learning methods for real-time decision making. RL learns from interaction with the environment via recognition and action to achieve a goal. On each interaction step, the agent (system) based on the state of the environment chooses an action that alters the state of the environment, and a reward or punishment is then provided to the agent as the desirability of the chosen action. In other words, the agent chooses an action based on policy, and the policy is learned through trial-and-error interactions of the agent with its environment. RL algorithms are very useful for solving a wide variety of problems especially when the model is not known in advance.

In RL the state can be a discrete function or continuous function. For environments with continuous state space, the Fuzzy RL has shown superiority as fuzzy inference emulates the human way of thinking and learning [18]. In RL a more intelligent agent, a human, defines the state space based on the available sensory input data. However, to address this problem, we build a state-action space based on a set of observed patterns of sensory input data and system output interactions (actions) with the environment without any previously given knowledge to the system.

In this work we propose a self-organizing hierarchical state-action space based on the input and output data of system which system use to interact with a dynamic environment where it is partially visible to the system. Then the system receives a set of feedbacks from environments which based on them it adjusts its internal structure. However, the feedbacks can be interpreted as another form

of input data, but in here we consider feedback as reinforcement data for simplicity, a distinguished form of data different than input and output data.

Related Works

All expert systems use a variation of learning methods depending on the problem they are solving. The common notion in all learning methods is to find a set of patterns which describes the problem space through them. For instance, in classifying images, the learning model learns the feature-patterns in input images for each category, or even for object recognition the system extracts and learns a set of feature-patterns for each trained object. In here we just focus on learning methods that adapt or modify themselves based on train data or some feedback, such as classification and RL. However, as in clustering the system forms set of clusters based on input data, but the method does not learn to adapt based on any feedback.

Reinforcement Learning

Standard RL theories are based on the concept of Markov decision process (MDP). A MDP is denoted as a tuple $\langle S, A, R, P \rangle$, where S is the state space, A is the action space, R is the reward function (feedback), and P is the state transition probability, Fig. 1.

The goal of RL is to learn the optimal policy π^*, so that the expected sum of discounted reward of each state will be maximized

$$J_{\pi^*} = \max_{\pi} J_{\pi} = \max_{\pi} E_{\pi}[\sum_{t=0}^{\infty} \gamma^t r_t] \quad (1)$$

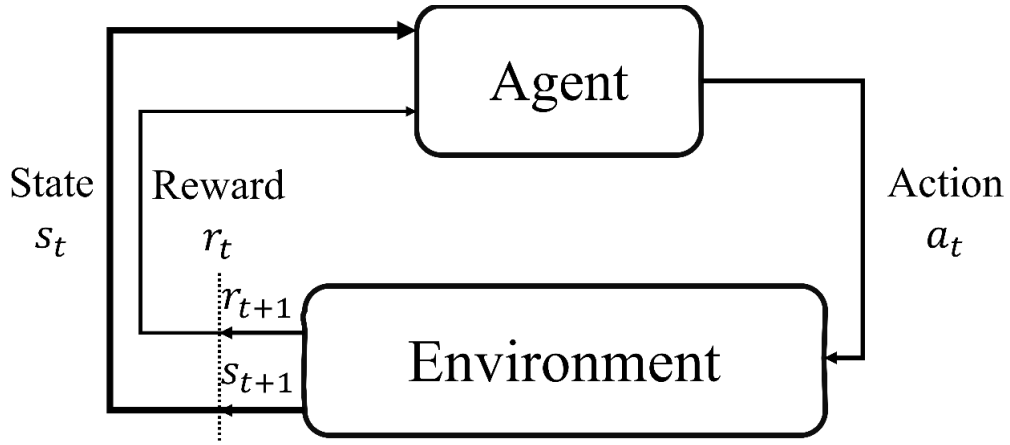

Fig. 1. Interaction of IS (agent) with environment with RL.

where $\gamma \epsilon [0,1)$ is the discount factor which if set to zero, it makes the agent "opportunistic" about current reward while agent with γ equal to 1 strives for a long-term high reward. r_t is the reward at time-step t, $E_\pi[.]$ stands for the expectation with respect to the policy π and the state transition probabilities, and J_π is the expected total reward. A value function $Q(s,a)$ represents the estimate of expected return attainable from executing action a in state s. Its computation repeatedly sweeps through the state-action space of MDP. The value function of each state-action pair is updated according to

$$Q(s,a) \leftarrow \sum_{s'} p(s'|s,a)\left[r(s,a,s') + \gamma \max_{a'} Q(s',a')\right] \quad (2)$$

until the largest change Δ in the value of any state-action pair is smaller than a preset constant threshold, where $p(s'|s,a)$ is the probability of state transition from s to s' after executing action a and $r(s'|s,a)$ is the corresponding reward. After the algorithm converges, the optimal policy is followed by simply taking the greedy action in each state s as

$$a^* = arg\max_a Q^*(s,a), \quad (\forall s \in S) \quad (3)$$

As for model-free cases where the agent has no prior knowledge of the environment, Q-learning (an RL algorithm) can achieve optimal policies from delayed rewards. At a certain time step t, the agent observes the state s_t, and then chooses an action a_t. After executing action a_t, the agent receives a reward r_{t+1} and gets into the next state s_{t+1}. Then the agent will choose the next action a_{t+1} according to the best-known knowledge and learned policy. Let α_t be the learning rate where α_t equal to zero makes the agent incapable of learning anything while α_t equal to one makes it

consider only the most recent information. The one-step updating rule of *Q*-learning can be described as,

$$Q(s_t, a_t) = (1 - \alpha_t)Q(s_t, a_t) + \alpha_t \left(r_{t+1} + \gamma \max_{a'} Q(s_{t+1}, a')\right) \quad (4)$$

Q-learning algorithm chooses the best action based on the state-action pair with highest *Q* value; however, in SARSA (State-Action Reward State-Action) [1] actions are chosen by ε-greedy policy and updating algorithm is described as follows,

$$Q_{t+1}(s_t,\ a_t) = (1 - \alpha_t)Q_t(s_t, a_t) + \alpha_t \left(r_{t+1} + \gamma Q'_{t+1}(s_t, a_t)\right) \quad (5)$$

where in (5) Q' is the *Q* value selected with ε-greedy policy at $t + 1$ for (s_t, a_t). In ε-greedy policy the action is chosen based on highest *Q* value with probability of 1-ε otherwise is chosen randomly, where $\varepsilon < 0.1$. ε-greedy policy allows the agent to explore the state-action space more and it avoids fast convergence to local optimal solution.

Incremental Learning Method for RL

Traditional RL techniques such as *Q*-learning and temporal difference (TD) algorithm [32] have focused on stationary tasks with fixed environment. To deal with the changing environment, an agent has to learn non-stationary knowledge incrementally and adapt to the new environment gradually. In [19], they propose an incremental learning method for RL in a dynamic environment. Their approach is designed to guide the previous optimal policy to a new one adapting to the new environment when the environment changes. First, a RL agent (e.g., *Q*-learning) computes the value functions and optimal policy in the original environment. When the environment changes, it generates a RL detector-agent to detect the changed part of environment (as called *drift environment*). Then it primarily updates the value functions for the drift environment and its neighboring environment using dynamic programming, which is called prioritized sweeping of drift environment. Finally, the agent starts a new RL process with the partly updated value functions to a new optimal policy adapting to the new environment, aiming at fusing new information (drift environment) into the existing knowledge system (previous optimal policy) in an incremental way.

RL learns knowledge from delayed rewards when interacting with the external environment. We assume that the environment changes when any changes of reward functions are detected.

When the environment changes, the objective for incremental learning in dynamic environments is to guide the original optimal policy π^* to a new one π_n^* that adapts to the new environment by fusing new information into the existing knowledge system incrementally.

Environment Drift

Environment drift refers to a change of the environment model over time. It implies that the corresponding reward r of a certain state-action pair (s, a) has changed over time. The drift environment is defined as the set of all state-action pairs whose rewards in the new environment differ from those in the original one.

Assume that $E(S, A, R, P)$ is the original environment model obtained by the RL agent and $E_n(S_n, A_n, R_n, P_n)$ is the new environment model that will be obtained. Given a state-action pair$(s, a), s \in S, a \in A, s \in S_n, a \in A_n$, if $r(s, a) \neq r_n(s, a), r \in R, r_n \in R_n$, which means that when taking action a in state s, the one-step reward $r_n(s, a)$ obtained in the new environment is different from the reward $r(s, a)$ obtained in the original environment, then the system gets the message that this given state-action pair (s, a) has drifted. The set of all state-action pairs which have drifted is called "drift environment", which can be formulated as $E_d(S_d, A_d, R_d, P_d)|r_n(s_d, a_d) \neq r(s_d, a_d)$, where $s_d \in S, a_d \in A, s_d \in S_n$ and $a_d \in A_n$.

Drift Detection

The first step for this algorithm is to detect any changes in the environment of reward for a pair of state-action. System needs to observe the rewards both in the original and the new environment to get the changed part. Since the original environment model E has been obtained by the RL agent, the only remaining thing is to observe rewards in the new environment. Due to having no prior knowledge about the new environment, the algorithm generates a detector-agent to explore it by executing a virtual RL process. The RL detector-agent observes the rewards by fully exploring the new environment with equal probability all the time.

Let us assume the total number of all state-action pairs in the original environment is N_{sum}, and the number of the traversed state-action pairs (also contained in the original environment) explored by the detector-agent is N_e. When $\frac{N_e}{N_{sum}} \geq \rho$, where ρ is a preset threshold close to but less than 1, the detector-agent ends the detection process for exploring the new environment. Finally, it

compares the rewards of state-action space in the new environment with those in the original environment, and obtains the changed part, i.e., drift environment.

Since the drift environment is the source of environmental changes, we can update the state-action space corresponding to drift environment by dynamic programming with top priority, using the value functions from the original environment and the rewards from the new environment. Then the value functions of the neighboring state-action space will also change under the influence of drift environment.

Integrated Incremental Learning Algorithm

First, in the original environment, the RL agent obtains the optimal policy π^*, value functions $Q(s,a)$ $(\forall s \in S, a \in A)$ and the environment model $E(S,A,R,P)$ through a standard Q-learning process.

Second, after any environment drift is detected, the new environment differs from the original environment in the form of rewards changing. The different part where the rewards have changed is denoted as drift environment $E_d(S_d,A_d,R_d,P_d)$, which generates new data in the new environment.

Third, compared with the original environment, the value functions of the drift part in the new environment tend to have the largest changes since drift environment is the source of new data. Then, the value functions of the neighboring environment will have relatively smaller changes on the influence of drift environment. Therefore, it gives priority to the m-degree neighboring environment of drift environment $E_{dn}^m(S_{dn},A_{dn},R_{dn},P_{dn})$ to sweep the value functions using dynamic programming.

Finally, it initializes the value functions of the new environment with the combination of the value functions of Q_{dn} and Q. The part of E_{dn}^m with prioritized sweeping stands for the new information. The part of the original environment E stands for existing knowledge to accelerate the learning process in the new environment. Based on this mechanism of fusing new information into the existing knowledge system, the agent restarts a standard Q-learning process and computes the new optimal policy π_n^* in an incremental way.

Ultimately, they have tested their algorithm on a changing maze. In general, their algorithm is proposed for optimizing the transporting goods inside Amazon warehouses.

Fuzzy Inference Systems and RL

For realistic agents, such as robots, drones, etc., which deal with continuous state-action space, the original *Q*-learning or other RL methods cannot be applied. In [18], [20], and [21], fuzzy RL has been applied to various problems with continuous state-action space, as fuzzy inference system (FIS) has proven to be extremely effective on problems for domains with continuous space such as control systems.

Q-Learning and FIS

In Fuzzy Inference System (FIS), we generate several if-then rules based on fuzzy membership functions. In FIS, we have several parameters that need to be tuned,

1) Number of fuzzy membership function per dimension of problem space and their parameters.
2) Number of fuzzy rules and each rule, the process of how to combine the right membership functions to generate the best rules.

Supervised learning method has been used to tune these two major parameters in FIS. However, for problems in control domain, we need to tune the system in real-time. To optimize the structure of fuzzy rules, we use the RL. As it is described in [21], FIS learning includes two phases, structural and parametric learning. Generally, the first phase consists in tuning the number of rules, i.e., the number of fuzzy labels per variable, while the second phase can be used to tune both the position of input and output fuzzy sets. They have developed a *Q*-learning method for FIS that finds and tunes the fuzzy rules and its parameters in real-time based on the feedback the system receives from the environment. They have tested their algorithm on various simulation and compared them with traditional FIS methods. Their algorithm out-performed others.

FIS and RL

In the previous section, we reviewed how online learning methods such as *Q*-Learning has improved real-time adaptation of fuzzy inference systems by finding the optimized parameters. As RL methods are applicable on problems with quantified state-action space then we can use FIS quality of mapping from continuous input space to continuous output space in RL for environments with continuous state-action space. For example in [18] we have a robot whose goal is to maintain

a specific distance to a wall and follow the walls of a room. To train a robot like that we need a neural network as depicted in Fig. 2. The first layer is the sensory-input layer, second layer is the fuzzification of input data (mapping of input to their relative membership functions) which each membership function is Gaussian function $\mu_{ij}(x_i) = exp\left[-\left(x_i - c_{ij}\right)^2/\sigma_{ij}^2\right], i = 1,2,\dots,r; j = 1,2,\dots,n$, where r is the number of input variables and for each we have created n membership functions. The inference system is based on Takagi-Sugeno method. In layer 3 the network combine the input from second layer (firing strength of each rule) as follow, $\Phi_j(x_i, x_2, \dots, x_r) = exp\left[-\sum_{i=1}^{r}\left(x_i - c_{ij}\right)^2/\sigma_{ij}^2\right], j = 1,2,\dots,n$. Normalization of neurons activity happens in layer 4 as follow, $\alpha_i = \frac{\Phi_i}{\sum_{j=1}^{n}\Phi_j}, i = 1,2,\dots,n$. Finally defuzzifications happens in layer 5 using the center-of-gravity method, the output variable as a weighted summation of incoming signals is given by $y = \sum_{j=1}^{n}\alpha_j o_j$, where y is the value of an output variable and o_j is the consequent of the jth rule. In their approach each rule R_i has m possible discrete actions $A = \{a_1, a_2, \dots, a_m\}$ and it memorizes the parameter vector Q associated with each of these actions. These Q-values are then used to select actions so as to maximize the discounted sum of the rewards obtained while achieving the task. They build the FIS with competing actions for each rule R_i as follows, $\boldsymbol{IF}\ X\ is\ S_i\ \boldsymbol{THEN}\ a_1\ with\ q(S_i, a_1)\ \boldsymbol{OR}\ a_2\ with\ q(S_i, a_2)\ \boldsymbol{OR}\ \dots \boldsymbol{OR}\ a_m\ with\ q(S_i, a_m)$. The continuous actions performed by the learner for a particular state is a weighted sum of the actions elected in the fired rules that describe the state, whose weights are normalized firing strengths vector of the rules, α. Subsequently we updated the *Q*-values of the elected actions according to their contributions.

Their proposed fuzzy *Q*-learning has three main features,

1. It is a fuzzy system with ellipsoidal regions of rules that are generated automatically.
2. Continuous actions can be generated. The continuous action is an average of the actions of every fuzzy rule weighted by the firing strengths of fuzzy rules and the *Q*-values of the actions are updated according to the contributions of fuzzy rules.
3. Prior knowledge can be embedded into the fuzzy rules, which can reduce the training time significantly.

In generation of continuous actions they use similar exploration-exploitation strategy for selecting the best action similar SARSA algorithm among all fuzzy rules. The system updates the *Q*-values similar to original *Q*-learning algorithm. However, in here they also update the fuzzy rules parameters dynamically. For example if for a new input data point the firing strength of some membership functions are less than a specific threshold then the system generates a new membership function with center fixed to value of data input and width set the maximum distance of this input data to its closest membership functions. Also, based on other criteria and the error of the system the center of fuzzy membership functions and their width will be updated accordingly.

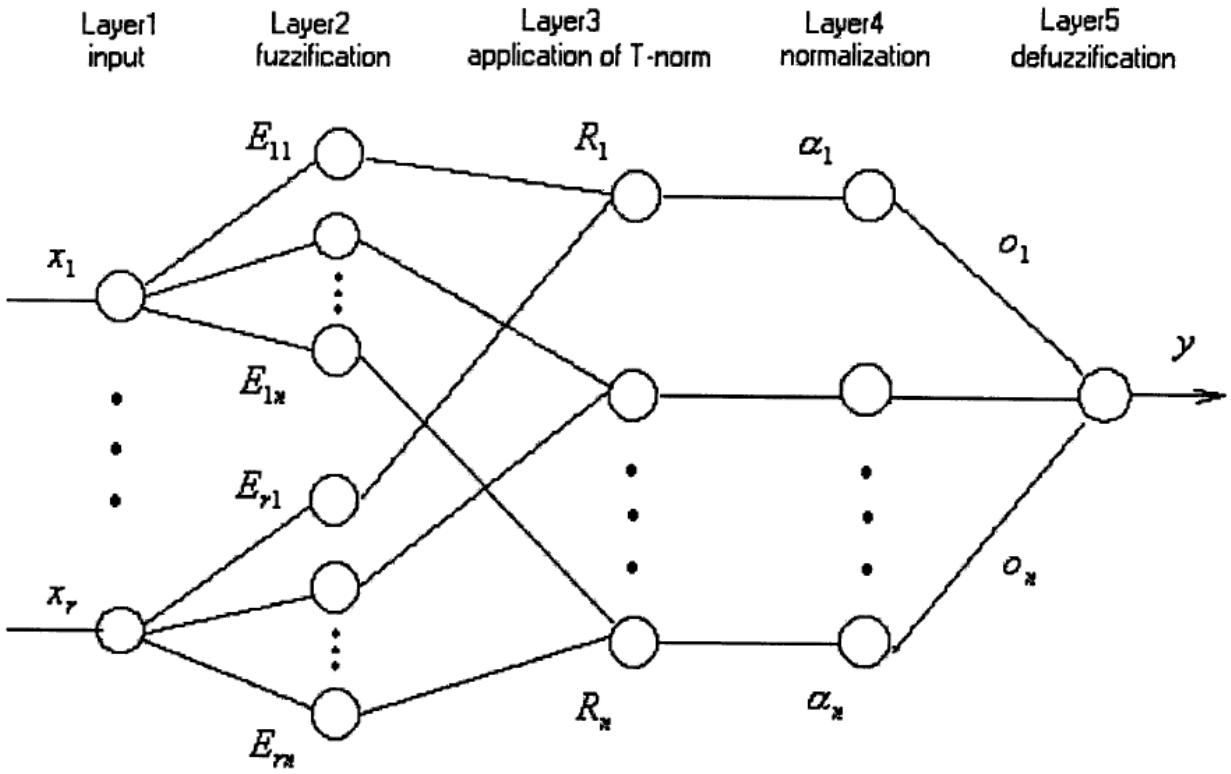

Fig. 2. Structure of fuzzy rule set for *Q*-Learning.

They tested their learning algorithm on a robot that must follow a wall in a room by keeping a constant distant to the wall. Then later after training the robot on a specific room they placed it in a new room with different interior design to observe the effect of prior learning on learning speed for the new environment which proved that their system adapt faster with prior knowledge in a new environment. In Fig. 3, we have the performance for an agent with a prior knowledge.

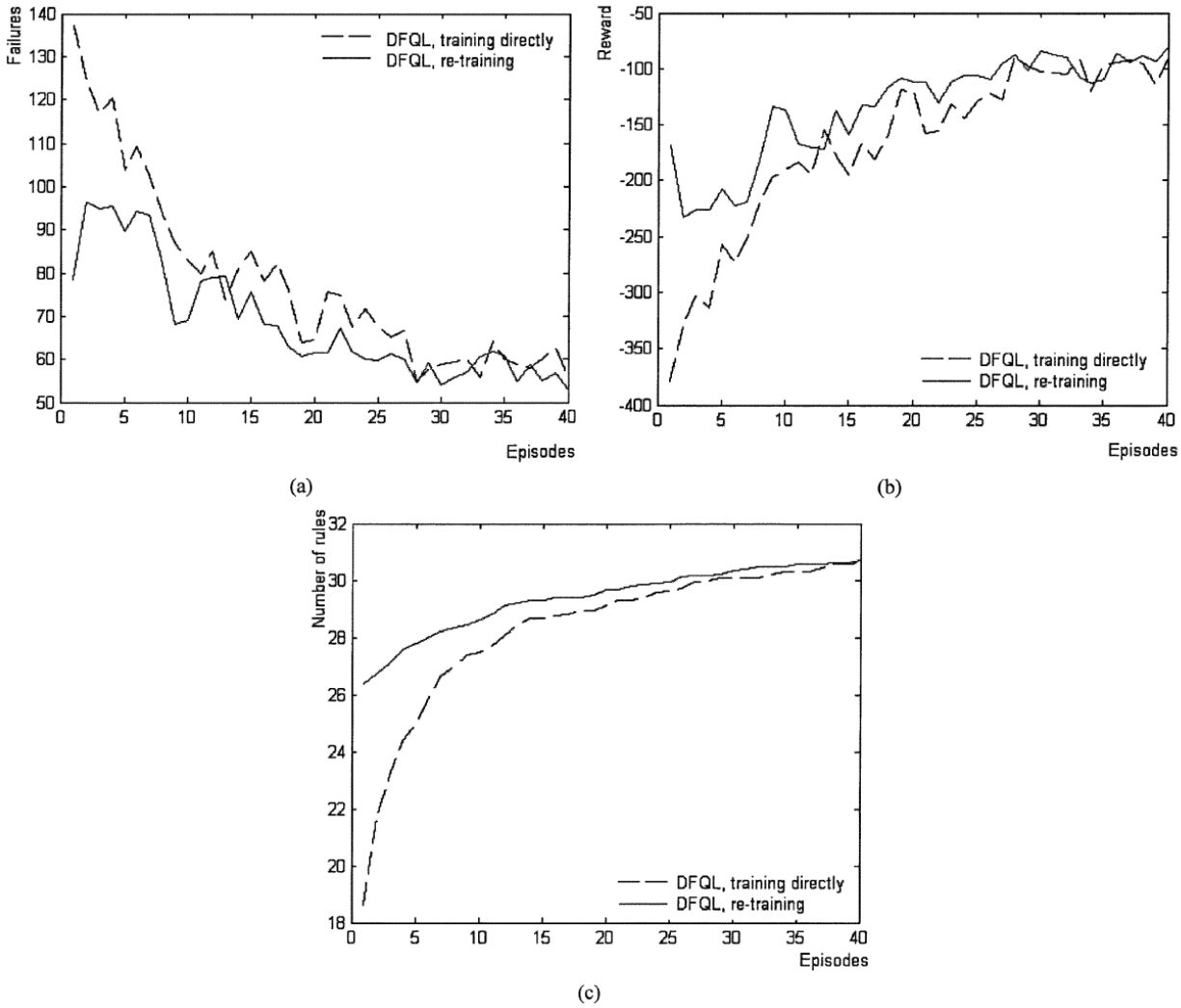

Fig. 3. Performance comparison of dynamic fuzzy Q-learning (DFQL) with training directly and retraining in a new environment. (a) Number of failures versus number of episodes. (b) Reward values versus number of episodes. (c) Number of rules generated versus number of episodes.

Deep Learning – Classification and RL

Deep learning (DL), inspired by the mechanism of human vision, recently attracted more and more attention due to its good performance in many fields such as speech recognition, computer vision, and natural language processing [2]–[4]. The intention of DL is to discover more abstract representations in higher levels [5]. It involves a class of models to hierarchically learn high-level features of input data with a deep hierarchical architecture. DL has the general formulation as follow,

$$f(\boldsymbol{x}) \approx g_1\left(g_2\left(\ldots\left(g_n(\boldsymbol{x})\right)\ldots\right)\right) \tag{6}$$

where $\boldsymbol{x}$ is the input, $g_i(i = 1, \ldots, n)$ is the operation on the ith layer, and $f(\boldsymbol{x})$ is the new representation of $\boldsymbol{x}$. The input of a higher layer is the output of its previous layer in the DL models. In this way, it can progressively lead to more abstract and complex features at deeper layers. More abstract features are generally invariant to most local changes of the input. Commonly used DL models include deep belief networks [6], deep Boltzmann machines [7], stacked auto-encoders (SAE) [8], and convolutional neural networks. Recent study also shows that deep models can give better approximations to nonlinear functions than shallow models [9], [10].

As DL has a hierarchical structure, thus it can be applied to any classical learning algorithm such as classification [22]-[25]. Also, most importantly recently DL has been applied to RL [26]-[27] on the transformation of state-action space as well as utilization of deep network to find the best action policy in a very dynamic environment such as games [28] where the agent played a game to master it after abundant iterations.

Playing Atari with Deep Reinforcement Learning

Learning to control agents directly from high-dimensional sensory inputs like vision and speech is one of the long-standing challenges of reinforcement learning (RL). Most successful RL applications that operate on these domains have relied on hand-crafted features combined with linear value functions or policy representations. Clearly, the performance of such systems heavily relies on the quality of the feature representation.

Recent advances in deep learning have made it possible to extract high-level features from raw sensory data, leading to breakthroughs in computer vision and speech recognition. These methods utilize a range of neural network architectures, including convolutional networks, multilayer perceptrons, restricted Boltzmann machines and recurrent neural networks, and have exploited both supervised and unsupervised learning. It seems natural to ask whether similar techniques could also be beneficial for RL with sensory data.

However reinforcement learning presents several challenges from a deep learning perspective. Firstly, most successful deep learning applications to date have required large amounts of hand labelled training data. RL algorithms, on the other hand, must be able to learn from a scalar reward signal that is frequently sparse, noisy and delayed. The delay between actions and resulting

rewards, which can be thousands of time-steps long, seems particularly daunting when compared to the direct association between inputs and targets found in supervised learning. Another issue is that most deep learning algorithms assume the data samples to be independent, while in reinforcement learning one typically encounters sequences of highly correlated states. Furthermore, in RL the data distribution changes as the algorithm learns new behaviors, which can be problematic for deep learning methods that assume a fixed underlying distribution.

They have demonstrated in [28] that a convolutional neural network can overcome these challenges to learn successful control policies from raw video data in complex RL environments. The network is trained with a variant of the *Q*-learning algorithm, with stochastic gradient descent to update the weights. To alleviate the problems of correlated data and non-stationary distributions, they use an experience replay mechanism which randomly samples previous transitions, and thereby smooth the training distribution over many past behaviors. They apply our approach to a range of Atari 2600 games implemented in The Arcade Learning Environment (ALE). Atari 2600 is a challenging RL testbed that presents agents with a high dimensional visual input (210×160 RGB video at 60Hz) and a diverse and interesting set of tasks that were designed to be difficult for humans players. Their goal is to create a single neural network agent that is able to successfully learn to play as many of the games as possible. The network was not provided with any game-specific information or hand-designed visual features, and was not privy to the internal state of the emulator; it learned from nothing but the video input, the reward and terminal signals, and the set of possible actions—just as a human player would. Furthermore the network architecture and all hyper-parameters used for training were kept constant across the games. So far the network has outperformed all previous RL algorithms on six of the seven games they have attempted and surpassed an expert human player on three of them.

Recent breakthroughs in computer vision and speech recognition have relied on efficiently training deep neural networks on very large training sets. The most successful approaches are trained directly from the raw inputs, using lightweight updates based on stochastic gradient descent. By feeding sufficient data into deep neural networks, it is often possible to learn better representations than handcrafted features [33]. These successes motivate our approach to reinforcement learning. Our goal is to connect a reinforcement learning algorithm to a deep neural network which operates directly on RGB images and efficiently process training data by using stochastic gradient updates.

Tesauro's TD-Gammon architecture provides a starting point for such an approach. This architecture updates the parameters of a network that estimates the value function, directly from on-policy samples of experience, $s_t, a_t, r_t, s_{t+1}, a_{a+1}$, drawn from the algorithm's interactions with the environment (or by self-play, in the case of backgammon). Since this approach was able to outperform the best human backgammon players 20 years ago, it is natural to wonder whether two decades of hardware improvements, coupled with modern deep neural network architectures and scalable RL algorithms might produce significant progress.

In contrast to TD-Gammon and similar online approaches, they utilize a technique known as experience replay [34] where they store the agent's experiences at each time-step, $e_t = (s_t, a_t, r_t, s_{t+1})$ in a data-set $D = e_1, \dots, e_N$, pooled over many episodes into a replay memory. During the inner loop of the algorithm, they applied Q-learning updates, or minibatch updates, to samples of experience, $e \sim D$, drawn at random from the pool of stored samples. After performing experience replay, the agent selects and executes an action according to an ε-greedy policy. Since using histories of arbitrary length as inputs to a neural network can be difficult, our Q-function instead works on fixed length representation of histories produced by a function ϕ.

Second, learning directly from consecutive samples is inefficient, due to the strong correlations between the samples; randomizing the samples breaks these correlations and therefore reduces the variance of the updates. Third, when learning on-policy the current parameters determine the next data sample that the parameters are trained on. For example, if the maximizing action is to move left then the training samples will be dominated by samples from the left-hand side; if the maximizing action then switches to the right then the training distribution will also switch. It is easy to see how unwanted feedback loops may arise and the parameters could get stuck in a poor local minimum, or even diverge catastrophically. By using experience replay the behavior distribution is averaged over many of its previous states, smoothing out learning and avoiding oscillations or divergence in the parameters. Note that when learning by experience replay, it is necessary to learn off-policy (because our current parameters are different to those used to generate the sample), which motivates the choice of Q-learning.

In practice, the algorithm only stores the last N experience tuples in the replay memory, and samples uniformly at random from D when performing updates. This approach is in some respects limited since the memory buffer does not differentiate important transitions and always overwrites

with recent transitions due to the finite memory size N. Similarly, the uniform sampling gives equal importance to all transitions in the replay memory. A more sophisticated sampling strategy might emphasize transitions from which we can learn the most, similar to prioritized sweeping.

In summary, they have evaluated their algorithm on seven Atari game and they use GPU to accelerate the computation and learning process. The system was able to learn the best policy after 100 epochs and each epoch is around 30 minutes of game play. Their work is a proof that with help of deep learning and RL methods we can tackle problems that previously deemed impossible to solve.

In [27], we have asynchronous method for deep RL to expedite the learning process. The goal of their research is to have several actors (agents) to explore the environment under various parameters and learn their own policies during an epoch and then by the end of the epoch they merge their learned parameters. They have extended their algorithm from 1-step Q-learning and SARSA method of n-step learning which they store feedbacks of n previous steps and adjust the network based on gradient descent to find the optimized policy faster in less training epochs.

Finally, as the first successful Artificial General Intelligence (AGI) [35], we have a deep and super deep neural network that is formed from several layers with several nodes per layer. Each node is a deep network on its own. The purpose of this super network is to learn various problems on various domains with evolutionary methods such as Genetic Algorithm (GA) to find the best learning path for each domain in the network. The network has shown great elasticity on learning continuously on new domains with previous knowledge learned on different domains.

Reinforcement Learning Architect with Self-Organized State-Action Space

GOL (Generic Online Learning), our proposed IS, must be able to adapt to any environment dynamic or static, as well as not having any previous knowledge of environment of action policy. GOL forms its abstract definition of state-action space with three hierarchical deep layered network consist of entities. In this research, we claim only three layer is sufficient for the purpose of online-learning. Each entity at each layer is the abstract definition of its lower layer entities.

Environment

As the environment is dynamic, the GOL (agent) perceives it via a set of sensory input channels. Also, the perception of the agent from the environment can be partial, some of the input data can be missing or corrupted. The input data enters the system as a stream in real-time.

Channels

Channels are the critical part of GOL structure, as it forms its perception of its surrounding environment based on the input data it receives and makes interaction with the environment via output channels.

Input Channel (IC)

Input channel is the source of input data stream to GOL which it perceives its environment through ICs. As an example, if the agent is a robot then visual and auditory data can be defined as ICs.

Output Channel (OC)

Output channel is the mediator between agent and environment for interaction. For our robot example above, its arms, fingers, and legs can be defined as OCs.

ICs input data format is a stream of numerical data, and OCs output is format is the same as ICs which OCs' output will be translated into an actual action based on the type of channel and the agent we are building.

GOL Entities

There are two types of entity.

1. Numerical Entity (EN).
2. Generic Entity (GN).

As the structure of GOL has three layers, the ENs form the first layer, and GN forms the other two. As entities in the second and third layer are similar in nature, we distinguish them into two

groups for clarity. We define GN of the second layer as *Medium-Entity* (EM) and for the third layer as *High-Entity* (EH).

Numerical Entity (NE)

NE is the representation of numerical value with some loss in the granularity of value, which EN can be a fuzzy number [11]-[12]. In this research, for simplicity instead of using fuzzy numbers to reduce the number of ENs, we do the conversion with an acceptable loss in the granularity of number up to two decimal to an EN. To clarify, numbers such as 102, 110, and 100 are mapped to one EN.

Medium Entity (EM)

EM represents a pattern of observed ENs for each channel over time. As in each layer, we store a limited number of entities then each EM is formed from the history of ENs.

Each EN has one-to-many connections to several EMs as those EM are built based on ENs in the first layer

High Entity (HE)

HE represents a high-level abstract pattern of observed EMs for a set of ICs and OCs over time. Each EH is formed from an EMs' history of some chosen channels. EH is the high-level description of observed input and output data over time. As granularity of EH is lower than EMs, then we expect to have less number of EHs compare to EMs.

Each EM has one-to-many connections to several EHs as those EH are built based on EMs in the second layer.

GOL Structure

GOL has a hierarchical structure with three layers made of two different types of entities.

GOL First Layer

First layer, where the input data and previous output data (actions made by the system) are entering the system as a stream. As the data are numerical data, then the system converts each input data to an EN.

GOL includes its previous executed actions (output data) because it finds the EH of the observed environment and its interaction with the environment to adapt to the best policy in time. Thus, EH describes the environment and GOL's actions in time for an adaptation of GOL.

First layer stores EN of IC and OC data stream for the past m_1 observation, where we consider $10 \leq m_1 \leq 50$ and in this research, we set m_1 to 50. So, when there is m_1 ENs in memory of each channel in the first-layer and new data is observed then GOL forget the oldest data. As Fig. 4 depicts the first layer, where each node represents an EN. In Fig. 4 new data stream enters the system from right side, so the old data are pushed to left side.

GOL Second Layer

Second layer, consists of EMs of each channel as a stream. As later we will describe the GOL learning algorithm, GOL finds the best four EMs set (4EM) matching the current data of each channel in the first layer, in the current time frame.

In Fig. 5 and Fig. 6 we will have at most four EMs per each channel in the first layer which they are input data to the second layer, Fig. 7.

In Fig. 7, for the second layer like the first layer, it stores the last m_2 observed data stream, which for second layer we set m_2 to 20. However, as it is evident from Fig. 7 the granularity of time for second layer is lower than first layer, as each 4EM set describe the pattern of data in first layer for a specific time-frame then history of these 4EMs that shapes second layer is representative in a different time frame, or more precisely a longer time-frame.

Also, according to Fig. 7, GOL perceives both sets of ICs and OCs as a high-level channel (HC) in the second layer. As in this layer input or output data has lost their meaning and they are both considered as input data.

GOL Third Layer

Third layer, is the highest layer in GOL where EHs are stored there, and this layer has no other functionality.

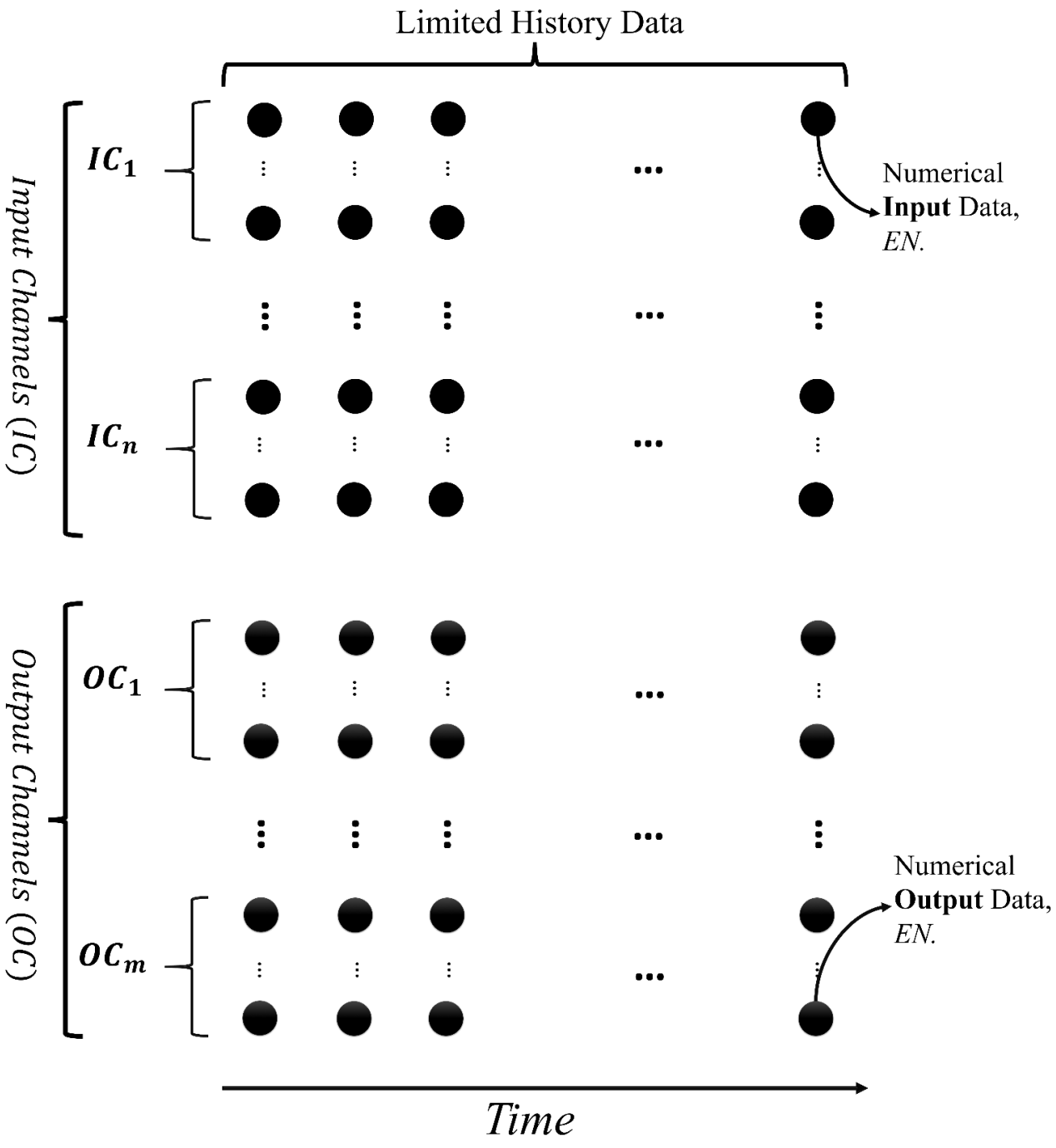

Fig. 4. First layer of GOL with limited memory for the data stream of OC and IC in the format of ENs.

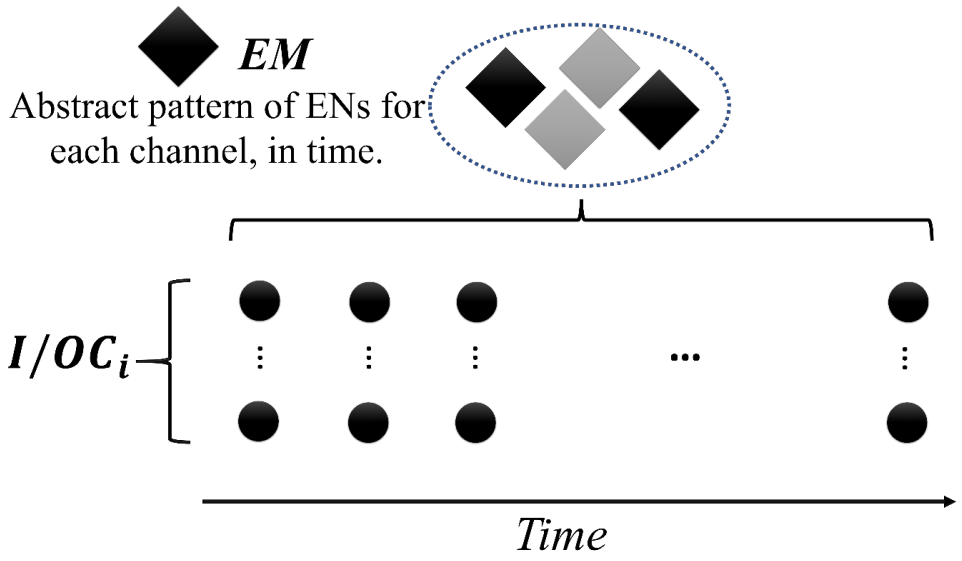

Fig. 5. EMs generated from ENs in the first layer or have high similarity with ENs in the first layer.

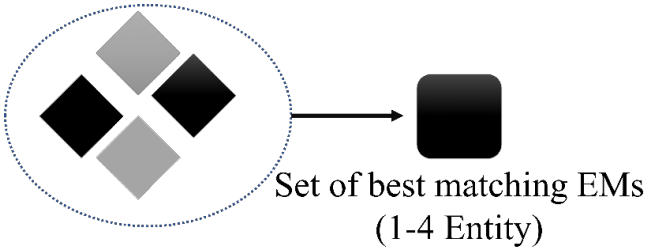

Fig. 6. A group of selected EMs based on ENs in the first layer as input data for the second layer.

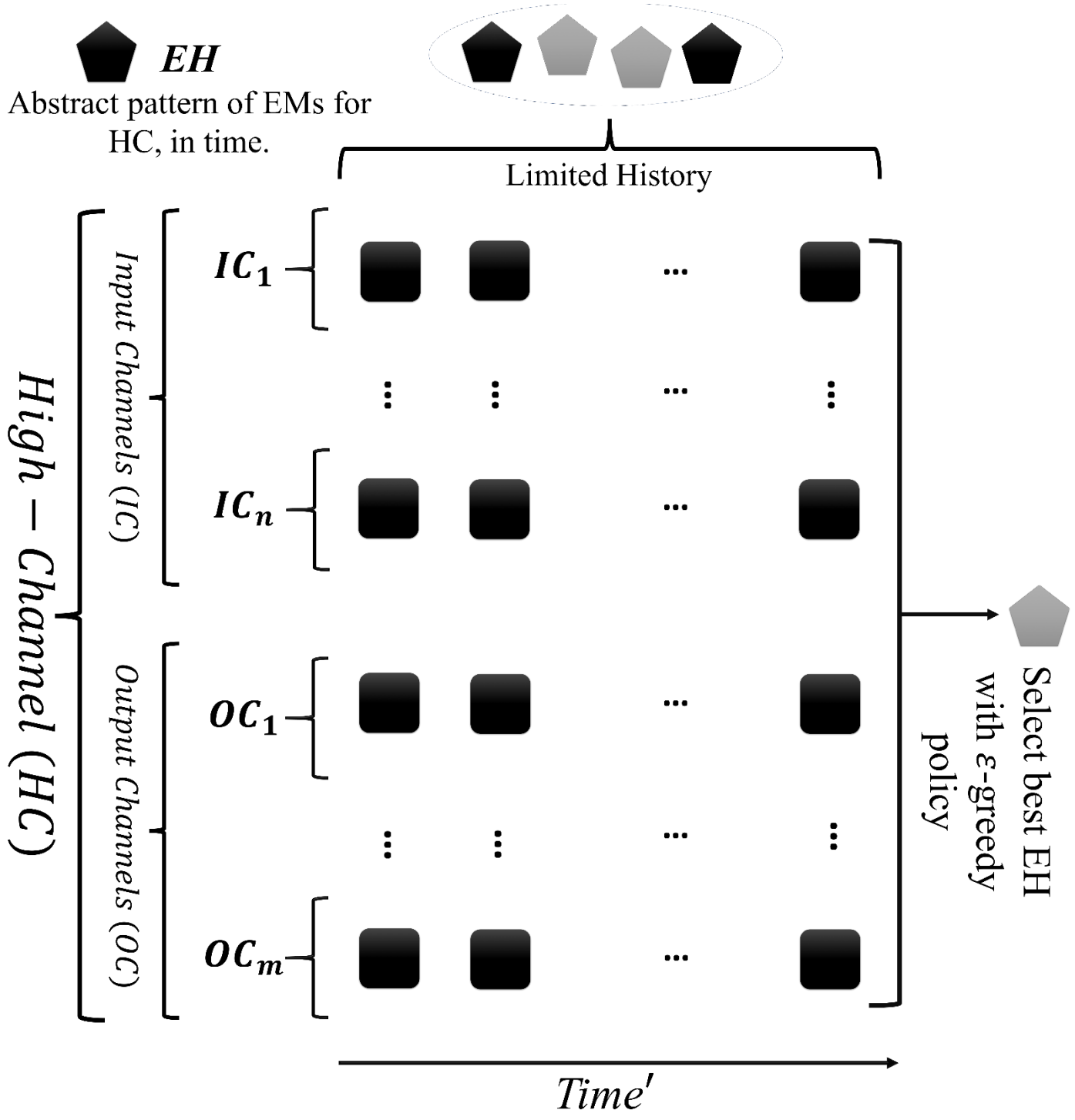

Fig. 7. Second layer of GOL.

GN Network

As it is described, EHs and EMs are formed in the first and the second layer based on their lower layer entities. Both EH and EM have the same general structure; however, we represent them separately for clarity.

EM Network

EM network, represents a pattern of ENs over time for an IC or OC. EM network is structured in three sections,

1. Past.
2. Present.
3. Future.

Fig. 8 depicts EM network structure in detail. Fig. 9, describes the purpose of each node in EM network. Each main node in EM network represents an entity in the lower layer. EM network is separated into three sections because the present section is used to match each EM with the current data in a channel of the first layer, the past section is used to validate the best matching EMs based on their present network entities, and the future section is used to predict the coming pattern or select best possible action.

According to Fig. 8, each entity-node (eN) in entity's network that represents an entity in a lower layer has an immediate connection to the eN in front of it in the next time step. Also, each eN has random connections with its neighbor nodes in the next time step in the network. For simplicity in this research, the connections between nodes have no extra data such as weight. Also, the structure of network will remain constant during learning. The eNs in the different section are connected to each other. Each section does not detach eN's connections in network structure, the main purpose of head-nodes and section-nodes, Fig. 9, are to organize the network and make evaluation and prediction computationally more convenient.

EH Network

EH network, represents HC. Thus it includes all EMs of selected ICs and OCs. In Fig. 10, we have EH network structure, as we notice its structure is similar to EM's network except the fact that each section has two head-nodes. One head-node for ICs' entities and another one for OC's entities; however, the connections between eNs of each part (IC and OC) exist as this entity describes the pattern for all input data and output data in a time-frame.

In EM and EH networks' structure, the number of layers in present section is higher than the other two sections. As in this research, the ratio is as follows, 3:6:1 (past : present : future), with majority given to present section.

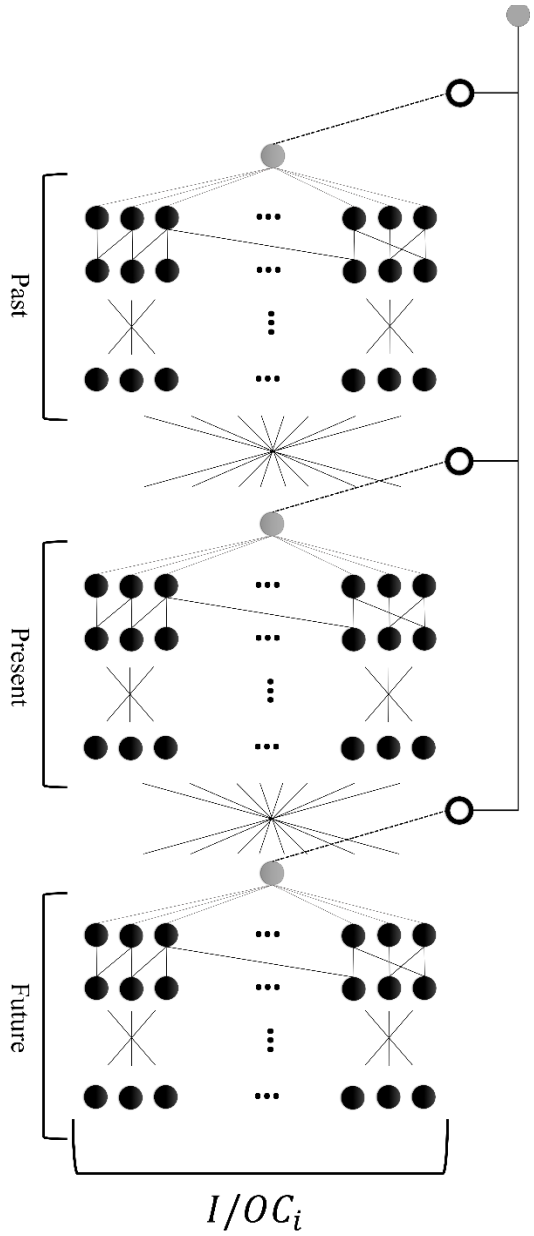

Fig. 8. EM network structure and its sections for an IC or OC.

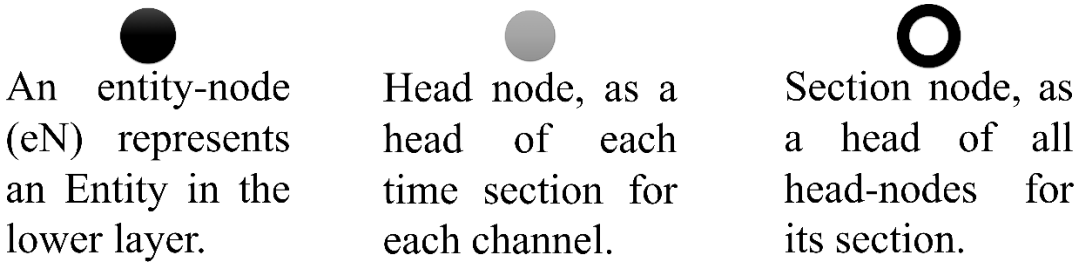

Fig. 9. EM network nodes description, including eNs and section nodes.

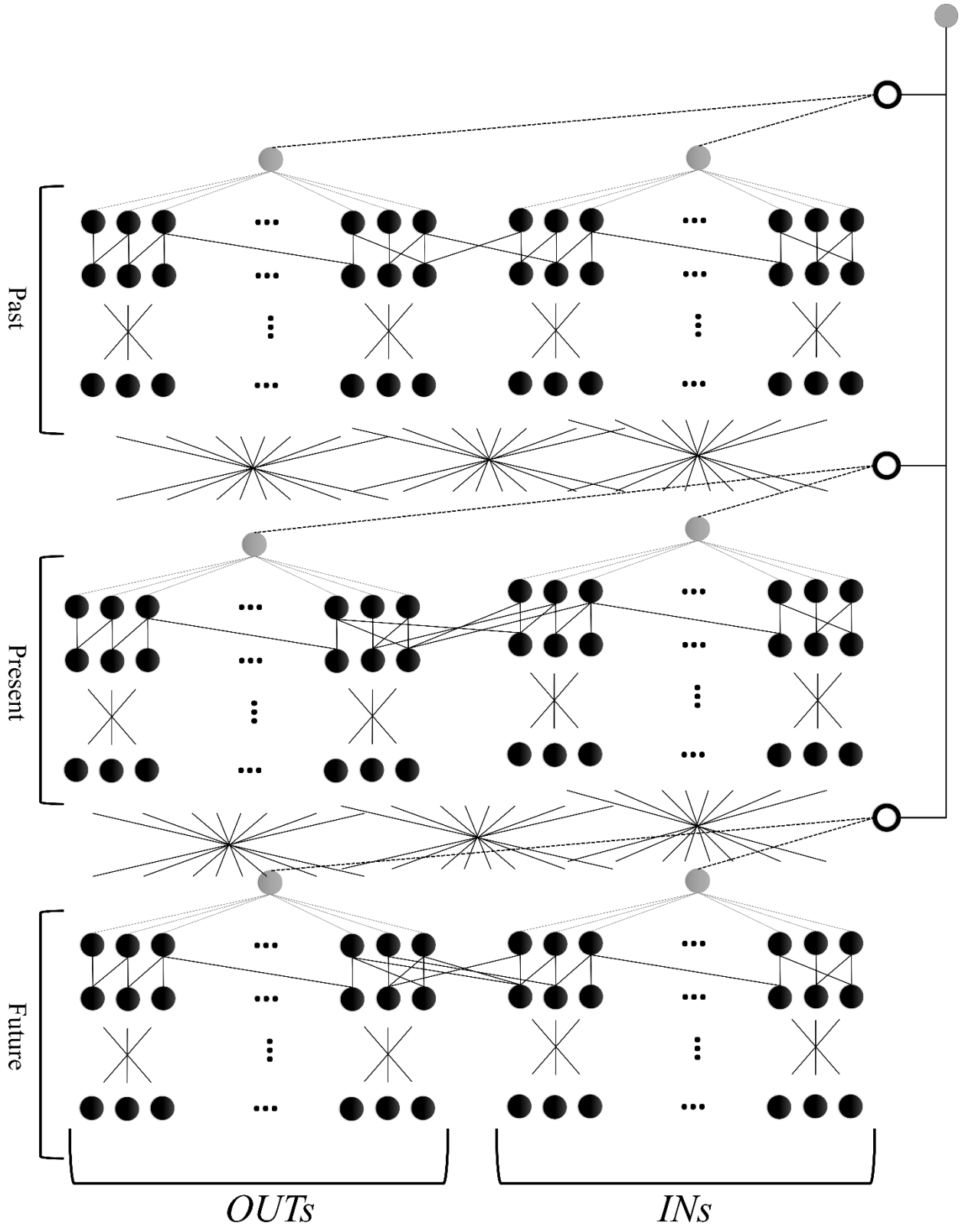

Fig. 10. EH network structure and its sections for selected ICs and OCs.

GOL Learning Algorithm

As GOL is an RL system, then it adapts based on received feedbacks for the actions it has executed. So, first GOL needs to find the best action based on the data it receives then adapts based on the observed feedbacks.

Find the Best EH

First GOL must find the best EH, which describes the recently observed data and self-taken actions, to select the best action. However, the system needs to have at least one EH to start making decisions; we call the system with no EH a crude system.

Input: ENs of the first layer.

1. A set of sensory input data from each channel arrives; however, some sensory data may be missing or corrupted.
2. GOL finds ENs of each input data and output data, of its previous actions, for each channel.
3. GOL adds the current ENs to the memory of first entities layer of **each channel**, first-layer (FML).
4. Get the list of all EMs (cEM) that current ENs defines them.
5. For every EM in cEM match their network with entities in FML for their respective channel. *Referring to Algorithm 2*.
6. Pick the best 4 possible matches based on ε-greedy policy for each channel (4EM) in cEM after the comparison.
7. Add 4EM of each channel to second layer (SML).
8. Find all the best matching EHs (cEH) based on the entities in second layer. *Referring to Algorithm 2*.
9. Pick the best EH (bEH) in cEH with ε-greedy policy as the best solutions or the observed patterns.
10. Make an action based on bEH.
    a. Pick all last EMs which they are OUT EMs in bEH.
    b. Now for each out-channel based on ε-greedy policy pick the best action.

Output: The best action for each OC.

* Value of $\boldsymbol{\varepsilon}$ is set to 0.15 for all algorithms in this research.

Fig. 11. Algorithm 1, finding of the best action based on ENs in first layer.

When GOL is crude, it means the first layer or/and second layer do not have enough data to form any EMs and consequently an EH. During crude state, the system makes random actions based on the nature of its OC.

Input: Entities of first or second layer.

1. IF system is crude THEN build an entity. Even IF system is refined THEN with small probability make a new entity.
2. Start from PRESENT section of the entity's network and level by level match the nodes' entities with entities in FML/SML level by level. Number of matches give us the activation of PRESENT section.
3. Validate the entity based on its PAST section, then calculate the final activation value of each entity as

   a=PRESENT activation

   a'=PAST activation

   IF $a' < a$ THEN sign=-1

   ELSE sign=1

   a_e = entity activation.

   $a_e = a \times 0.7 + a' \times sign \times 0.3$
4. Select the 4 of the best matching entities based on ε-greedy policy.

Output: A list of matching entities, EMs or EHs, depending on the given layer.

Fig. 12. Algorithm 2, the finding of the best matching higher entities based on the lower layers.

When the first EH is formed, then it is refined to make decisions and adapt. In a period of crudeness, the first layer forms new EMs based on ENs, if no matching exists for the observed ENs. However, even during refined state and even existence of an EM with a good match with ENs in the first layer, the system forms a new EM with a small probability of 0.005. Also, the system has the same behavior in forming EH is the second layer but the probability of forming a new EH is 0.0005.

In Fig. 11, we have the algorithm of finding the best action based on the current ENs in the first layer. In Fig. 12 we have, algorithm 2, the algorithm to find the best EH based on entities in the lower layer.

GOL stores the last 10 EHs and their ENs that resulted in action, which they were the result of algorithm 1. Afterward, it uses them in adaptation step, Fig. 13.

Adaptation

After finding the best action for each OC, the system interacts with the environment. Simultaneously, the system is receiving feedbacks from the environment; however, the system does not know which executed action should receive a particular feedback as feedbacks are arriving with various delays.

Input: Stream of feedbacks.

- GOL collects reinforcement feedbacks it receives from environment between arrival of new data from channels.
- GOL store EHs from *algorithm 1* in a limited-sized list of LEH.

1. For each feedback (fb) in list of collected feedback.
   a. fb affects all entities in LEH (with size of $L = 10$) with effect-ratio ($\boldsymbol{er}$) er_k where k is the index of entity in LEH. In other words, k is the ranked position of action-entity (EH), which k equal 0 is the most recent action taken.
   b. er_k can follow one of the following equations,
      1) Linear ($er_k = 1 - k/L$)
      2) Exponential ($er_k = e^{-k^2}$)
      Where $er_k \leq 1$ for all $k \geq 0$, and in here we select linear equation for simplicity. Each out-channel and each portion of that channel can have a distinguish adaptation policy.
   c. Adaptation follows SARSA algorithm, which at the time of adaptation system finds the best action based on current data and uses (5) with α equal to 0.15 and γ equal to 0.85.

Fig. 13. GOL adaptation algorithm based on feedbacks.

Feedback can be another form of input data, but in here we consider them as a reinforcement data, not an input data, which some feedbacks are not given to system directly and are observable through input channels, they are indirect feedbacks which we ignore them in this research.

In Fig. 13, we have the adaptation algorithm based on the feedbacks and SARSA adaptation algorithm. After applying the current feedback for adaptation of GOL structure, the system keeps the current feedbacks with a probability of 0.2 to use them in the next adaptation iteration.

Simulation of the 5G Mobile Network Load-Balancer with Dynamic Parameters as the Environment

Provisioning of a high data rate wireless connectivity in rural and remote areas has became a stringent challenge for perspective 5G wireless networks. Unlike the previous generation, 5G mobile network is service driven rather than technology driven [13]. The concept of the "broadband access everywhere" is a pillar of 5G service requirements, and the target performance of 50 Mbps everywhere is considered regarding the experienced user throughput [14].

In this section, we apply GOL to event load balancer for the 5G mobile network; however, GOL, in general, can be applied to any cloud load balancer. There have been various approaches to network load-balancing issue [29]-[31]. For example in [30], with the help of knowledge of social network and inner-network formed by mobile users in the same local network they have proposed an optimal solution to effectively cache the most demanding data in the edge network and user devices. However, in this research, we are focusing on intelligently balance the load of allocation between various available cells in the network in real-time.

Events

Any mobile user constantly generates new events, such as.

A. Attach request: and event to register user device (UE) with a tower.
B. Tracking update: an event to notify the tower and cell about geo-location movement and movement between cells that cover different frequencies.
C. Service request: an event from UE to the tower that contains a request message for some data.

D. Paging request: an event from tower to Internet to fetch the requested data. In this research we do not consider this event as it is not generated by mobile user.
E. Handover request: an event that handover a UE to another tower.
F. Detach request: an event for termination of the connection with a tower.

User

In our simulation, each user has a unique ID called userID. Also, we have four types of users, premium-mobile, premium-stationary, basic-mobile, and basic-stationary. The number of premium users is smaller than the number of normal users.

Load Balancer (LB)

LB is responsible for assigning each arrival event per user to a virtual machine (VM) to store the event data. The size of event data is 2Kbytes. LB keeps track of assigned VM per user based on userID, frequency per user-type, frequency per event-type, and frequency per pair of (user-type, event-type). Fig. 14 depicts the LB and VMs.

Virtual Machine (VM)

In our simulation, we have *R* VMs, as we set *R* to three. Each VM cache-size (Cache-Size) can store only 700 user's data, Fig. 15. Each user type has its separate data cache-queue; the size of each queue can change in time as new users are arriving. Also, when there are 700 users stored in the VM and a new one arrives then VM removes the oldest one from one of the queues randomly. As new user data arrives, the old ones are pushed down the queue.

Cloud Storage (CS)

Cloud storage, according to Fig. 14, is used for data persistency in VMs, as old data in each user-queue may get deleted then each VM periodically synchronize each user data with CS based on user-data synchronization-time (ST) to have the most recent data in the cloud. In each VM as new users are arriving the old user-data are push down to the bottom of the queue. Thus VM reduces their ST for faster synchronization as the chance of deletion increases for the user-data near the bottom of each user-queue.

ICs and OCs for LB

LB uses GOL to find the best possible VM for each event. Thus, first we need to define the ICs and OCs for our load-balancing problem, and there are as follow,

$$IC_0 = \begin{cases} frequency\ of\ observing\ this\ user_{type} \\ frequency\ of\ observing\ this\ event_{type} \\ frequency\ of\ observing\ (user_{type}, event_{type}) \\ \substack{user_{type} \\ event_{type}} \end{cases} \quad (6)$$

$$IC_i = \begin{cases} queue\ size\ for\ each\ user_{type}\ in\ VM_i \\ average\ of\ ST\ for\ each\ queue\ in\ VM_i \end{cases} \quad 1 \le i \le R \quad (7)$$

$$OC_j = \begin{cases} ST\ for\ each\ queue\ in\ VM_j \\ value\ of\ VM_j\ (Q\ value) \end{cases} \quad 1 \le j \le R \quad (8)$$

However, in our simulation, we randomly set values of some input data in ICs except IC_0 to -1 to make the environment partially invisible to the LB.

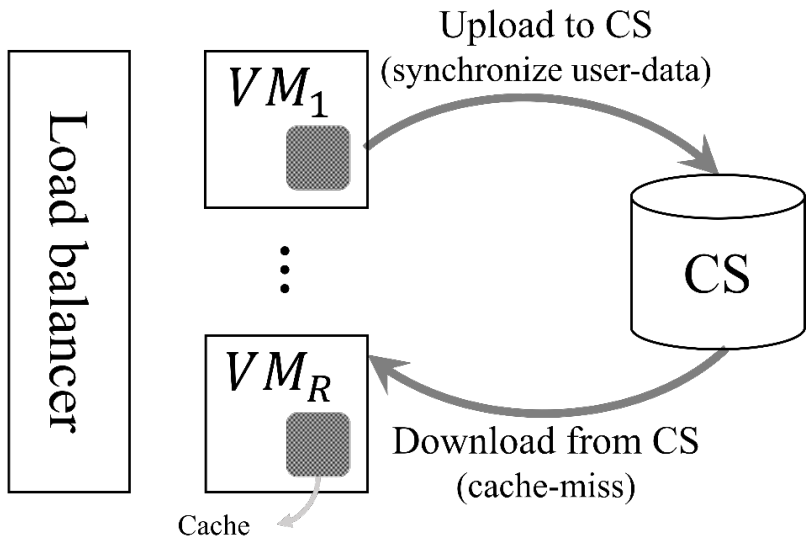

Fig. 14. The structure of LB and *R* number of VMs with CS for persistency of data.

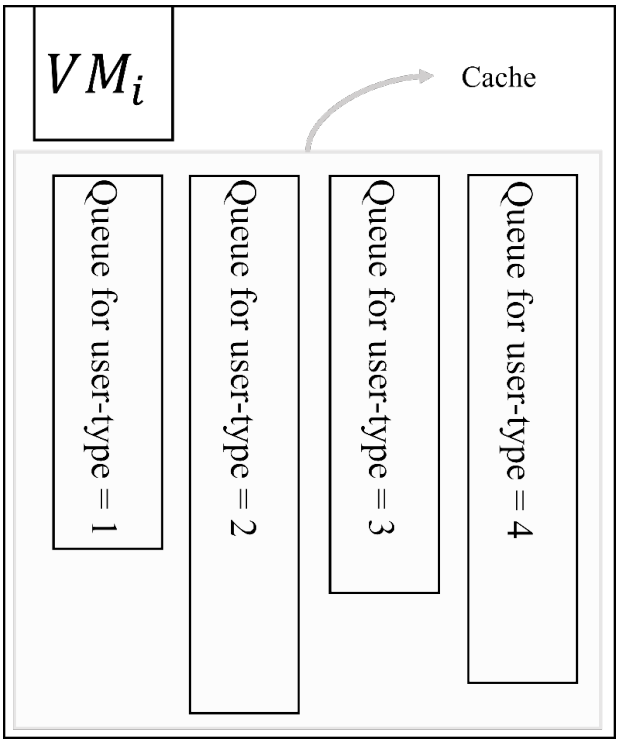

Fig. 15. The internal structure of a VM and user-queues.

Definition of Punishment, Reward, and System Objective

The LB has two objectives:

1. The first objective is to have minimum total cache-miss for all VMs, as LB tries to assign each user-data to a right VM with less probability of deletion of that user-data in time. As deletion of a user-data happens when they are pushed down in a cache-queue by insertion of new users into cache-queue of the VM.
2. The second objective is to have the minimum communication load between CS and VMs. When a cache-miss happens, the assigned VM checks the CS to download the latest stored data for the assigned user from CS. Also, a VM synchronize (upload data to CS) each user-data based on user's ST.

For GOL to minimize the cache-miss and communication load between VMs and CS, we need to define punishment and reward for each case, and they are as follow,

Punishment

I. *Cache-miss punishment* affects the value of selected VM (8). For each action entity (EM) we calculate the cache-miss punishment as follow,

$$p_{cm}^{i} = -er_k \times (100 - 10 \times u_t) \quad (9)$$

where p_{cm}^{i} is the cache-miss punishment of action taken for ith VM in (7) (jth VM in (8)), u_t is the user-type and for $u_t = 1$ which means top priority user (premium-mobile) receives the maximum punishment for a cache-miss, and er_k is described in Fig. 13.

II. *Cloud-load punishment*,

Upload

When cloud-load is type of *upload* to CS, then it influences ST of user-data with the following punishment,

$$p_{cu}^{i} = -er_k \times (5 \times ds) \quad (10)$$

where p_{cu}^{i} is the upload punishment due to synchronization of user-data with CS, and ds is the upload data in bytes which is 2KB.

Download

When the type of cloud-load is *download* from CS, then we have a cache-miss and punishment is defined as,

$$p_{cd}^{i} = -er_k \times ds \quad (11)$$

where p_{cd}^{i} is the download punishment and ds is the download size in bytes which if CS has the data of user then ds is 2KB, otherwise it is the load of network communication with the CS.

III. *VM miss-match*, happens when LB assigns a new incoming event of allocated user to a different VM. Thus the system has an inner VM load. The punishment for inner-load is defined as follow,

$$p_{lv}^{i} = -er_k \times (0.1ds) \quad (12)$$

where p_{lv}^{i} is inner-VMs communication load due to miss-match.

Reward

System receives rewards for each event that it processes and they are as follow,

I. *VM value reward* is as follow,

$$r = 200 - 3 \times (n_{fb} - n_{ST}) \quad (13)$$

$$r_v^i = er_i \times \left(r_{rnd2} + \frac{r}{2}\right) \quad (14)$$

where r_v^i is the reward for VM value of executed action n_{fb} is number of feedbacks that GOL has received between processing two events, as feedbacks are delayed then the system can receive more than one feedback between arrival two user-events, n_{ST} is the number of upload punishments that effected ST of user-data, and r_{rnd2} is a random value between 0 to $r/2$.

II. *VM ST reward* for each user-type is defined as follow,

$$r' = cm - 10 \times n_{ST} \quad (15)$$

$$r_{ST}^{i} = er_i \times \left(r'_{rnd2} + \frac{r'}{2}\right) \quad (16)$$

where r_{ST}^{i} is the reward of synchronization time of users in all queues to increase synchronization time to reduce uploads to CS, and r'_{rnd2} is similar to r_{rnd2} in (14). In (15), cm is equal to 1000 and the intuition for selection of its value is that we estimated that n_{ST} could have a maximum value of 100, which during our experiment we observe that value of n_{ST} fluctuates between 3 to 5.

As the system receives feedbacks between processing each two events, it stores them then after processing all feedbacks with a probability of 0.2 it keeps all currently observed feedbacks for processing in the iteration due to nature of feedbacks that are delayed feedbacks.

Experiments and Results

In our simulation, we have the following parameters, in table I, for our environment.

TABLE I. ENVIRONMENT SIMULATION PARAMETERS

Parameter	**Description**
N_{users}	Number of mobile user in environment is set to **3000**.
$User_{type}$	Number of user-types is set **4** with 1 as premium user.
$Event_{type}$	Number of event-types is set to **5**.
R	Number of VMs is set to **3**.
Cache-Size	Maximum number of users that each VM can cache their data is set to **700** users.
N_{events}	Number of events is set to 50,000 events for entire simulation.

We evaluate performance of GOL for load balancing with following methods,

Memorized-BestVM (MB)

Memorized-BestVM (MB), for each new user to the system LB selects the best VMs based on the cache-queue size depending on the user-type and select its ST randomly from range of [300-15000] milliseconds (ms) and memorized where the data of this user is stored. Afterward, for all events related to stored users, it sends them to the same VM but each time chooses their ST randomly. Thus, in this case, we never have miss-matches in VMs, but we have cache-misses.

Memorized-BestVM with Fixed ST (MB-F)

MB-F is like MB but the ST always is set to 1000ms (1 second).

In our simulation, each epoch (time-frame) of simulation contains processing of 20 user-events. Table II depicts the result of our simulation for all three methods.

In Fig. 16, we have the comparison of our three load-balancing methods for average of cache-miss for three VMs during the entire simulation.

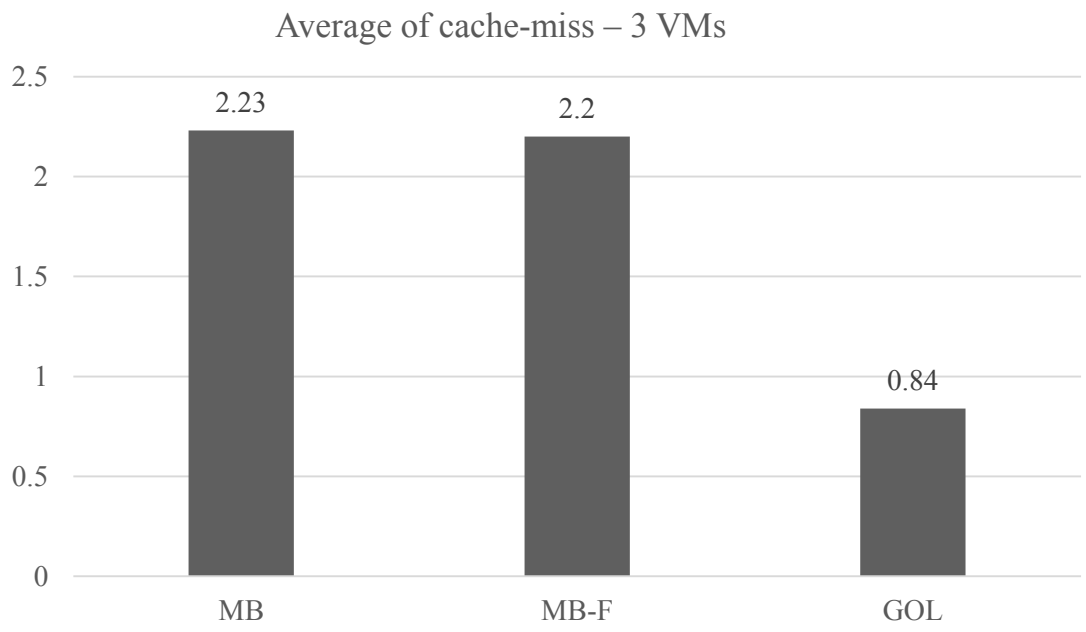

Fig. 16. Comparison for average of cache-miss.

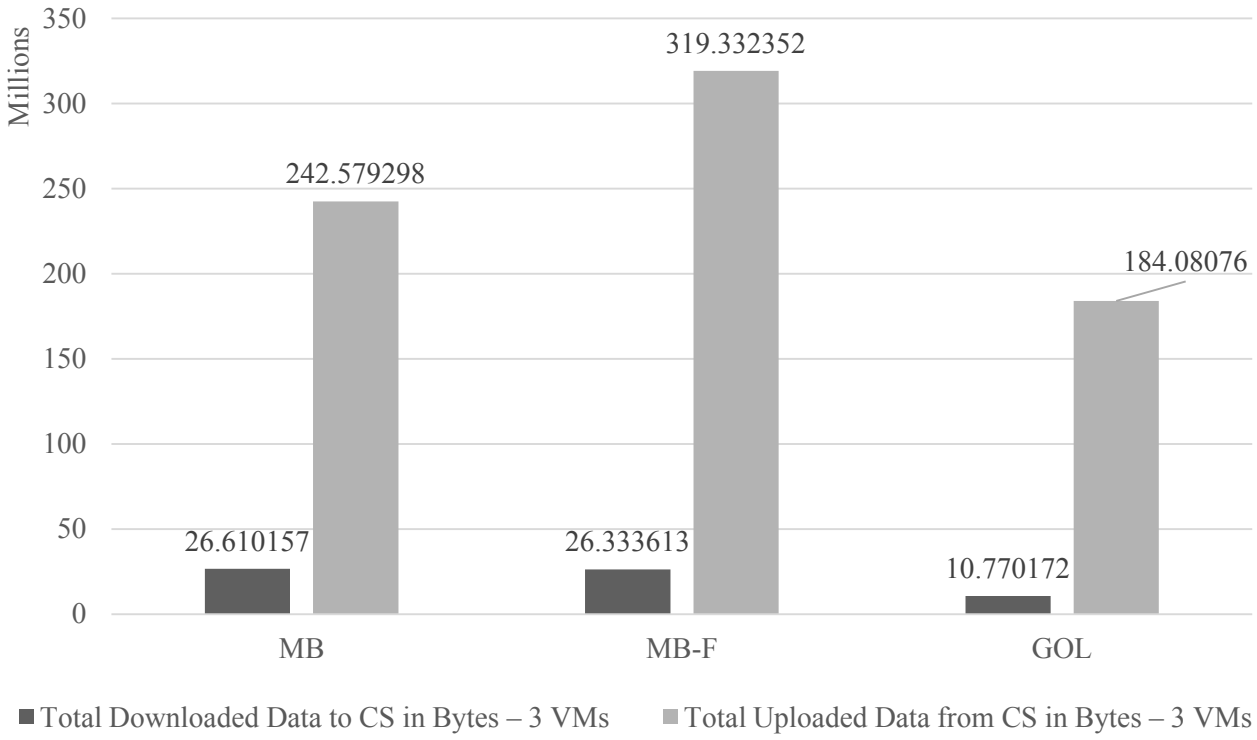

Fig. 17. Comparison for total communication load between VMs and CS.

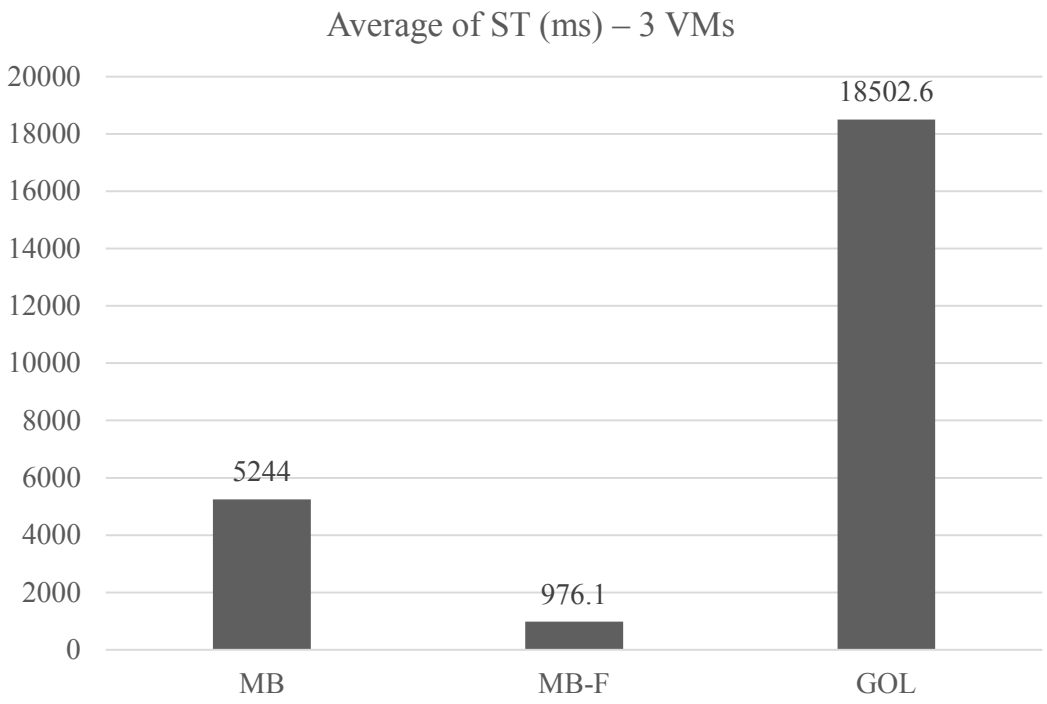

Fig. 18. Comparison for average of ST in three VMs in milliseconds.

In Fig. 17, we have the comparison of our three load-balancing methods for total communication load between VMs and CS. In Fig. 18, we have the comparison of our three load-balancing methods for average of ST for three VMs during the entire simulation.

TABLE II. RESULTS OF SIMULATION OF ALL THREE LOAD BALANCING METHODS

	Average of cache-miss – 3 VMs	Total Downloaded Data to CS in Bytes – 3 VMs	Total Uploaded Data from CS in Bytes – 3 VMs	Average of ST (ms) – 3 VMs
MB	2.23	26610157	242579298	5244
MB-F	2.2	26333613	319332352	976.1
GOL	**0.84**	**10770172**	**184080760**	**18502.6**

As we can notice from Table II, GOL has better performance, which confirms that the static structure of GOL with three layers has the capability to adapt to our environments without having previous knowledge and pre-defined state-action space. Figure 19 and 20 depict the growth of cache-miss for MB and GOL as the best two solutions for the load-balancing problem.

As we notice in Fig. 19 and 20, the growth rate of cache-miss with GOL is tremendously lower compare to MB. In Fig. 21-23 we have the increase in number of entities over time, which by the end of simulation we have a total of 51 EHs, 1747 EMs, and 8750012 ENs. As an EH is an abstract description of the observed input and output data over time, thus the number of EH represents the total number of patterns to describe the whole environment and system's interaction with that environment. As we may have over 8 million ENs, which some of them may have low usage frequency and can be removed but the total number of EHs is less than 100. Number of EHs proves that system has a good performance on finding the high-level pattern that describes a dynamic and

chaotic environment due to various randomness in the environment, also in feedback policies (14) and (16).

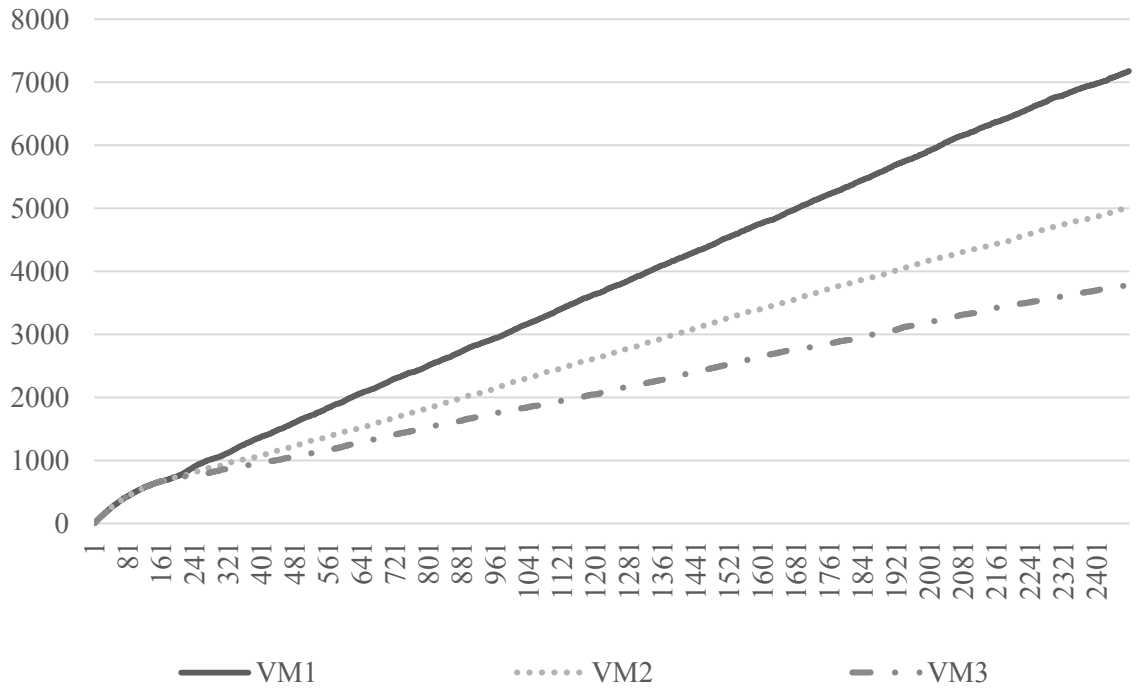

Fig. 19. Total cache-miss in time for MB (epochs).

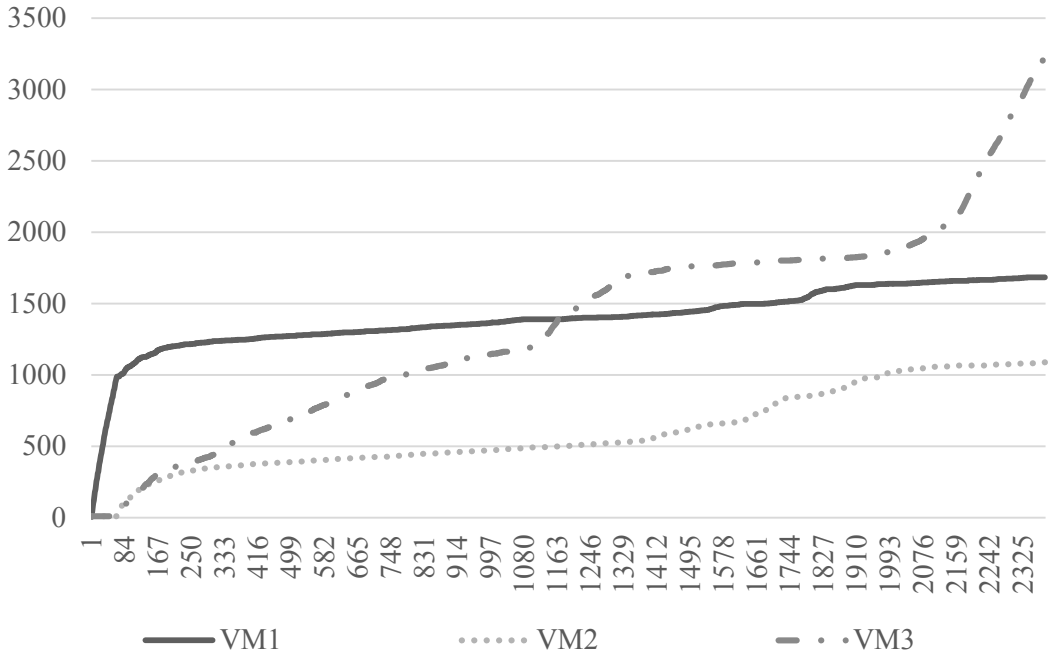

Fig. 20. Total cache-miss for GOL in time (epochs).

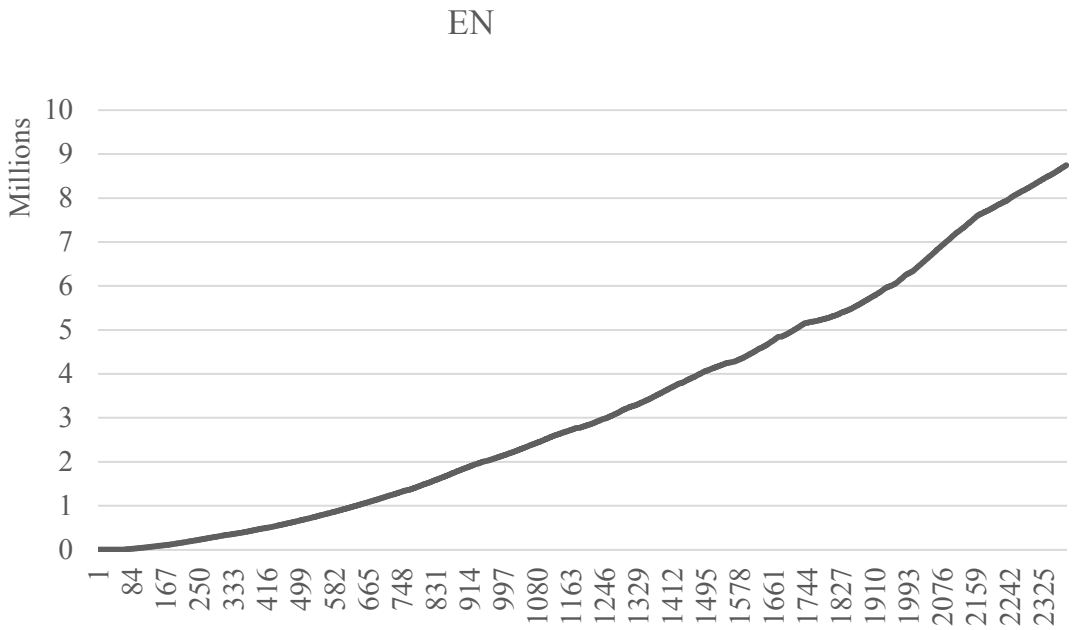

Fig. 21. Growth for number ENs over time (epochs).

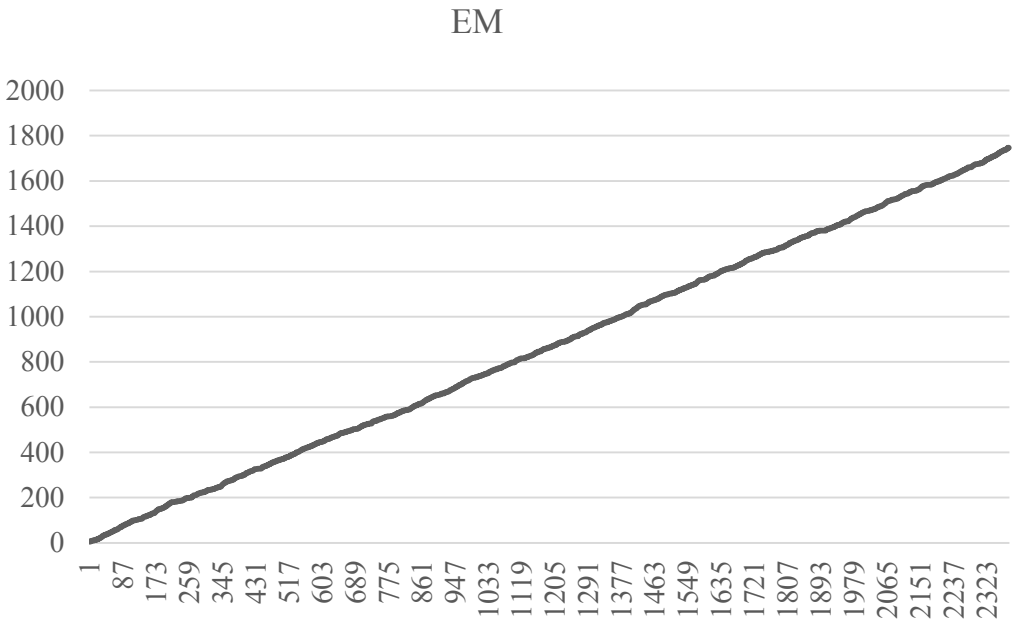

Fig. 22. Growth for number EMs over time (epochs).

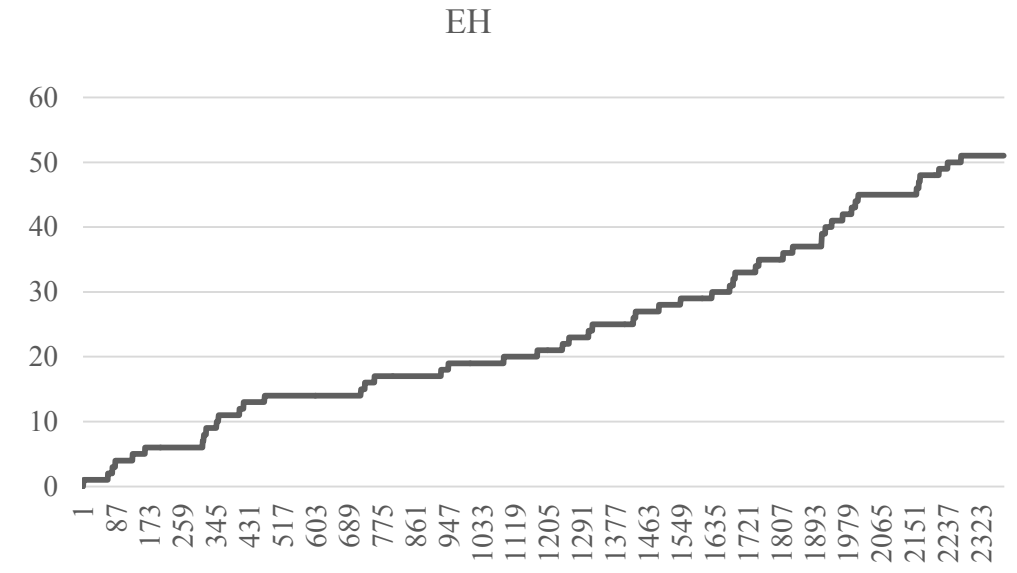

Fig. 23. Growth for number EHs over time (epochs).

Conclusion and Future Work

We have proposed an intelligent system based on RL algorithm with simple three deep-layered structure for a dynamic and partially visible environment where feedbacks are delayed. Then we tested our proposed method for load balancing in the 5G mobile network. In [15]–[17] RL has been used to solve various problems related to mobile networking and messaging. Our result for load balancing demonstrates that our proposed method has the capability to adapt to some optimal solutions over time just by forming high-level and abstract features from observed data in time.

However, there are several improvements that can be applied,

1. The growth rate of ENs is extremely high which can be reduced by using fuzzy numbers in the first layer to find the best matching EN for numerical input/output data.
2. Using Fuzzy Inference System (FIS), similar to fuzzy RL to determine the final action based on all final OUT EMs in the selected EH.
3. In here for simplicity the connections between nodes in entities network is random, and they are static; however, the connection between nodes can adapt in time to form a better structure.
4. Also, the connections between nodes in entities structure have no meaning which by adding some data such as weight we can give modify the activation procedure of each entity and finding the best possible actions.
5. In here to find the best matching entity based on data in lower layer we consider similarity layer by layer in entity network; however, a sub-graph of the network may not have any similarity with the current data. We can add a new sub-graph for that portion with a probability to avoid creating a new entity.

In this research, EH is built from all output channels and input channels; however, an EH may only be built from some selected ICs and OCs. In this way, different EHs belong to various contexts. The advantage of having context allows the system to have the fourth layer that connects similar EHs from different contexts based on their activation patterns and network structure, which ultimately these connections between different contexts can lead to new and different actions in time.

References

[1] R. S. Sutton and A. G. Barto, “Reinforcement Learning: An Introduction”, MIT Press, 1998.

[2] G. Hinton and R. Salakhutdinov, “Reducing the dimensionality of data with neural networks”, Science, vol. 313, no. 5786, pp. 504–507, Jul. 2006.

[3] A. Krizhevsky, I. Sutskever, and G. E. Hinton, “Imagenet classification with deep convolutional neural networks”, in Proc. Adv. Neural Inf. Process. Syst. 25, Red Hook, NY, USA, pp. 1097–1105, 2012.

[4] C. Farabet, C. Couprie, L. Najman, and Y. LeCun, “Learning hierarchical features for scene labeling”, IEEE Trans. Pattern Anal. Mach. Intell., vol. 35, no. 8, pp. 1915–1929, Aug. 2013.

[5] Y. Bengio and O. Delalleau, “On the expressive power of deep architectures”, in Algorithmic Learning Theory. Berlin, Germany: Springer, pp. 18–36, 2011.

[6] G. E. Hinton, S. Osindero, and Y. Teh, “A fast learning algorithm for deep belief nets”, Neural Comput., vol. 18, no. 7, pp. 1527–1554, Jul. 2006.

[7] R. Salakhutdinov and G. E. Hinton, “Deep Boltzmann machines”, in Proc. Int. Conf. AI Statist., Clearwater, FL, USA, pp. 448–455, 2009.

[8] Y. Bengio, P. Lamblin, D. Popovici, and H. Larochelle, “Greedy layerwise training of deep networks”, in Proc. Adv. Neural Inf. Process. Syst. 19. Cambridge, MA, USA, pp. 153–160, 2007.

[9] I. Sutskever and G. E. Hinton, “Deep, narrow sigmoid belief networks are universal approximators”, Neural Comput., vol. 20, no. 11, pp. 2629–2636, Nov. 2008.

[10] N. Le Roux and Y. Bengio, “Deep belief networks are compact universal approximators”, Neural Comput., vol. 22, no. 8, pp. 2192–2207, Aug. 2010.

[11] Peter Sussner; Estevão Esmi; Laécio C. Barros, “Controlling the width of the sum of interactive fuzzy numbers with applications to fuzzy initial value problems”, IEEE International Conference on Fuzzy Systems (FUZZ-IEEE), pp. 1453–1460, 2016.

[12] Piotr Prokopowicz, “Analysis of the changes in processes using the Kosinski's Fuzzy Numbers”, Federated Conference on Computer Science and Information Systems (FedCSIS), pp. 121–128, 2016.

[13] Jongtae Song; Taewhan Yoo; Pyung Jung Song, “Mobility level management for 5G network”, International Conference on Information and Communication Technology Convergence (ICTC), pp. 940–943, 2016.

[14] Alexander Karlsson; Osama Al-Saadeh; Anton Gusarov; Renuka Venkata Ramani Challa; Sibel Tombaz; Ki Won Sung, “Energy-efficient 5G deployment in rural areas”, IEEE 12th International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob), pp. 1–7, 2016.

[15] Pablo Muñoz; Raquel Barco; José María Ruiz-Avilés; Isabel de la Bandera; Alejandro Aguilar, “Fuzzy Rule-Based Reinforcement Learning for Load Balancing Techniques in Enterprise LTE Femtocells”, IEEE Transactions on Vehicular Technology, pp. 1962–1973, vol. 62, 2013.

[16] Behrooz Shahriari; Melody Moh, “Intelligent mobile messaging for urban networks: Adaptive intelligent mobile messaging based on reinforcement learning”, The 12th IEEE International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob), pp.1–8, 2016.

[17] Jin Wu; Jing Liu; Zhangpeng Huang; Shuqiang Zheng, “Dynamic Fuzzy *Q*-Learning for Handover Parameters Optimization in 5G multi-tier networks”, International Conference on Wireless Communications & Signal Processing (WCSP) , pp. 1–5, 2015

[18] Meng Joo Er; Chang Deng, “Online tuning of fuzzy inference systems using dynamic fuzzy Q-learning”, IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics), pp. 1478–1489, vol. 34, 2004.

[19] Zhi Wang; Chunlin Chen; Han-Xiong Li; Daoyi Dong; Tzyh-Jong Tarn, “A novel incremental learning scheme for reinforcement learning in dynamic environments”, 12th World Congress on Intelligent Control and Automation (WCICA), pp. 2426–2431, 2016.

[20] Chia-Feng Juang; Chia-Hung Hsu, “Reinforcement Ant Optimized Fuzzy Controller for Mobile-Robot Wall-Following Control”, IEEE Transactions on Industrial Electronics, pp. 3931–3940, vol. 56, 2009.

[21] L. Jouffe, “Fuzzy Inference System Learning by Reinforcement Methods”, IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews), pp. 338–355, vol. 28, 1998.

[22] Dawei Wang; Wei Ding, “A Hierarchical Pattern Learning Framework for Forecasting Extreme Weather Events”, IEEE International Conference on Data Mining, pp. 1021–1026, 2015.

[23] Yicong Zhou; Yantao Wei, “Learning Hierarchical Spectral–Spatial Features for Hyperspectral Image Classification”, IEEE Transactions on Cybernetics, pp. 1667–1678, vol. 46, 2016.

[24] Khanittha Phurattanaprapin; Punyaphol Horata, “Extended hierarchical extreme learning machine with multilayer perceptron”, 13th International Joint Conference on Computer Science and Software, pp. 1–5, 2016.

[25] Yanyun Qu; Li Lin; Fumin Shen; Chang Lu; Yang Wu; Yuan Xie; Dacheng Tao, “Joint Hierarchical Category Structure Learning and Large-Scale Image Classification”, IEEE Transactions on Image Processing, 2016.

[26] Wenzhi Zhao; Shihong Du, “Spectral–Spatial Feature Extraction for Hyperspectral Image Classification: A Dimension Reduction and Deep Learning Approach”, IEEE Transactions on Geoscience and Remote Sensing, pp. 4544–4554, vol. 54, 2016.

[27] Volodymyr Mnih; Adrià Puigdomènech Badia; Mehdi Mirza; Alex Graves; Tim Harley; , “Asynchronous Methods for Deep Reinforcement Learning”, Google Deep Mind, 2016.

[28] Volodymyr Mnih; Koray Kavukcuoglu; David Silver; Alex Graves; Ioannis Antonoglou; Daan Wierstra; Martin Riedmiller, “Playing Atari with Deep Reinforcement Learning”, Deep Mind, 2013.

[29] Ekram Hossain; Mehdi Rasti; Hina Tabassum; Amr Abdelnasser, “Evolution Toward 5G Multi-Tier Cellular Wireless Networks: An Interference Management Perspective”, IEEE Wireless Communications, pp. 118–127, vol. 21, 2014.

[30] Ejder Bastug; Mehdi Bennis; Mérouane Debbah, “Living on The Edge: The Role of Proactive Caching in 5G Wireless Networks”, IEEE Communications Magazine, pp. 82–89, vol. 52, 2014.

[31] Hisham Elshaer; Federico Boccardi; Mischa Dohler; Ralf Irmer, “Load & Backhaul Aware Decoupled Downlink/Uplink Access in 5G Systems”, IEEE International Conference on Communications (ICC), pp. 5380–5385, 2015.

[32] R. Sutton, “Learning to predict by the methods of temporal difference,” Machine Learning, vol.3, pp.9-44, 1988.

[33] Alex Krizhevsky, Ilya Sutskever, and Geoff Hinton. “Imagenet classification with deep convolutional neural networks”, In Advances in Neural Information Processing Systems 25, pp. 1106–1114, 2012.

[34] Long-Ji Lin. Reinforcement learning for robots using neural networks. Technical report, DTIC Document, 1993.

[35] Chrisantha Fernando, Dylan Banarse, Charles Blundell, Yori Zwols, David Ha, Andrei A. Rusu, Alexander Pritzel, Daan Wierstra, “PathNet: Evolution Channels Gradient Descent in Super Neural Networks”, Jan. 2017.