Q LEARNING REGRESSION NEURAL NETWORK

M. Sarıgül[*], *M. Avcı*[†]

Abstract: In this work, a Nadaraya-Watson kernel based learning system which owns general regression neural network topology is adapted to Q learning method to evaluate a quick and efficient action selection policy for reinforcement learning problems. By means of the proposed method Q value function is generalized and learning speed of Q agent is accelerated. The training data of the developed neural network are obtained by a standard Q learning agent on closed-loop simulation system. The efficiency of the proposed method is tested on popular reinforcement learning benchmarks and its performance is compared with other popular regression methods and Q-learning utilized methods. QLRNN increased the learning performance and it learns faster than other methods on selected benchmarks. Test results showed the efficiency and the importance of the proposed network.

Key words: *reinforcement learning, q learning, q value function approximation, general regression neural network, kernel based regression*

Received: December 07, 2016 **DOI:** 10.14311/NNW.2018.28.023
Revised and accepted: October 16, 2018

1. Introduction

Reinforcement learning is a branch of machine learning that is interested in finding an optimal policy that must be followed in transitions on certain states to maximize the total amount of rewards as a result of the selected actions [1]. Q-Learning is one of the most popular and successful reinforcement learning methods [2]. Since the large state space problems have a huge amount of state-action pairs, Q learning is not so effective in this kind real-world problems. This is called as "curse of dimensionality" by Sutton [3]. In the mentioned case regression-based approaches have been utilized. This led many regression methods to be adapted in reinforcement learning.

The most common methods used for generalization are artificial neural networks. Neural networks have a wide range of uses such as modeling network traffic [4, 5], predictive control for burning zone temperature [6], wind speed prediction [7], modeling time series [8] and even modeling of flue aimed temperature of

[*]Mehmet Sarıgül – Corresponding author; Iskenderun Technical University in Hatay, Computer Engineering Department, Turkey, E-mail: mehmet.sarigul@iste.edu.tr

[†]Mutlu Avcı; Cukurova University in Adana, Biomedical Engineering Department, Turkey E-mail: mavci@cu.edu.tr

coke oven [9]. These structures have also been successfully used in the field of reinforcement learning. Anderson used one-layer and two-layer networks with error backpropagation to generalize value function and showed his own two-layer network was able to solve pole-balancing task for TD learning [10]. Boyan and Moore concerned about the hardness of robust generalization of value function with a function approximator such as a neural network [11]. They proposed Grow-Support to prevent bad convergence in Dynamic Programming. Sutton suggested sparse-coarse-coded function approximators (CMACs) to avoid poor performance [3]. Bradke and Barto defined Least Square TD and Recursive Least Square TD algorithms and they also defined a recursive version of LS TD algorithm [12]. They also provided convergence proofs of the algorithms. Boyan updated LSTD algorithm in a way giving the ability to work with eligibility traces and named it as LSTD(λ) [13]. Brafman and Tennenholtz proposed a model based algorithm called RMAX [14]. Lagoudakis and Parr proposed Least Squares Policy Iteration method which able to learn state-action value function for Q learning [15]. Ernst et al. suggested offline working Fitted Q Iteration algorithm [16]. After completing each episode, obtained values are used in the regression algorithm. State-action values obtained by experience are kept in four-tuples form as (s, a, r, s_{t+1}), where, s is the transition state, a is the selected action, r is reward and s_{t+1} is the resulting state of action a. Riedmiller, taking the inheritance of Fitted Q algorithm, proposed NFQ algorithm representing Q-value function with a multilayer perceptron network [17]. In this algorithm experienced values are kept in triple form as (s, a, s_{t+1}), where, s is the transition state, a is the selected action and s_{t+1} is the resulting state of action a. Bonarini et al. proposed LEAP algorithm [18]. Puddle-world, mountain-car and cart-pole problems were solved by using LEAP algorithm as given in Dutech et al. [19]. XAI (eXplore and Allocate, Incrementally) is also a kernel based regression method proposed by Langlois to be used in reinforcement learning [19]. Whiteson and Stone proposed NeuroEvoulotion of Augmenting Topologies (NEAT) to approximate Q value function. NEAT-Q involves backpropagation to estimate the value function [20]. Heinen and Engel suggested supposed IPNN(Incremental Probabilistic Neural Network) by inspiring PNN and GRNN [21]. They applied it for regression on reinforcement learning problems [22]. Recently, the use of artificial neural networks in this area has increased even more as the popularity of deep learning structures has increased [23, 24].

In this paper, proposed QLRNN is a Q learning adaptation approach with Nadaraya-Watson kernel regression method on GRNN topology. Recently it has become very popular to use artificial neural networks as a function approximator with reinforcement learning methods. GRNN is a well-known and highly effective neural network structure. QLRNN has been developed to use this success for reinforcement learning problems. It is aimed to find an effective action selection policy rapidly for reinforcement learning problems. A standard Q agent runs on the environment and collects expected values of state-action pairs. While obtaining target values according to the existing experience of Q-agent, recent target values are stored in the neurons of the pattern layer of the network. After a number of episodes efficiency of the network is measured with the same problem. As QLRNN is an instance-based learning system, it can learn incrementally without an extra training phase. Test results show that QLRNN is able to find an appropriate

action selection policy quickly. Performance of QLRNN is compared with the popular methods such as LEAP(Learning Entities Adaptive Partitioning), NEAT-Q (NeuroEvolution of Augmenting Topologies), RMAX, LSPI (Least Squares Policy Iteration), XAI (eXplore and Allocate, Incrementally) and NFQ (Neural Fitted Q Iteration) on the selected benchmarks. QLRNN contains major differences according to the compared algorithms. LEAP algorithm divides the state-action space into pieces called macrostates. Each macrostate is responsible for a particular area and these areas do not overlap. In QLRNN, each neuron has a central point. The effect of a neuron at a point in the space is determined by the kernel function. The fields can be overlapped. This led to a better generalization over the state-action space. The NEAT-Q algorithm is suitable for network learning with back-propagation. QLRNN uses a kernel-based approach with one-pass learning which is a much faster approach then repetitive way. However, RMAX is a result-oriented successful algorithm, but it is not possible to use it for large-scale problems because of probability computation complexity for all states. Since QLRNN can represent many states in space with a single neuron, it can be offered as a solution to more complicated problems. LSPI offers an iterative learning approach by using policy iteration. LSPI trains a number of basis functions with sample data generated by following a specific policy. The type and number of functions are determined intuitively according to the problem. Even in the iterations, the type and function of the functions can be changed. This makes the use of LSPI more complicated. On the contrary, the kernel structure of QLRNN is specific. When QLRNN is used, the number of neurons in the network is first determined and no changes are needed throughout the training period. XAI is also a kernel-based method uses gradient descent to train Gaussian kernels. XAI does not have a predefined structure like QLRNN and needs an extra training procedure. QLRNN has also one-pass learning advantage over XAI. NFQ uses a multi-layer perceptron neural network trained with Rprop algorithm, a version of back-propagation. QLRNN is different from NFQ in structure and training. Also, QLRNN has one-pass learning advantage with respect to NFQ. QLRNN accelerates learning performance and it learns faster than other popular methods on selected benchmarks. Test results prove the efficiency and show the importance of the proposed network.

In the second part, fundamental approaches for QLRNN are explained in detail. QLRNN topology and QLRNN algorithm are given in the third part. The fourth section which is devoted to the test results and discussion is followed by the last part, conclusion.

2. Fundamental approaches for QLRNN

QLRNN is a Q learning adaptation method using Nadaraya-Watson kernel on GRNN topology to find an efficient action selection policy rapidly. Q learning algorithm is updated to be able to establish the pattern layer of the proposed neural network to be used generalizing Q value function.

2.1 Q-Learning

One of the most important breakthroughs in reinforcement learning was the development of Q-learning by Watkins [2]. In Q learning, the updating process is done on the action values. Best action of the following state is used as the return expectation in the update process. The update process of one-step Q-learning is done according to the Eq. 1.

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha[r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)], \tag{1}$$

where $Q(s_t, a_t)$ is value of the current action. $\max_a Q(s_{t+1}, a)$ is value of the following best action. r_{t+1} is the reward obtained in transition from s_t to s_{t+1}. α is a constant that specifies action value changing rate. γ is another constant specifying the effect of the next best action value on that of the previous one.

Algorithm 1 Q-learning algoritm.

Set all $Q(s, a)$ randomly
Repeat (for each episode):
Set s as one of the initial states
Repeat (for each step of an episode)
Select action a according to state s using policy derived from Q
Take action a, observe r, next state s_{t+1}
$Q(s_t, a_t) = Q(s_t, a_t) + \alpha[r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)]$
$s \leftarrow s_{t+1}$
Until s is terminal

In Q-learning algorithm for each episode, selected action values are updated according to Eq. 1. After finding a successful policy, the evolution process is completed. Q learning algorithm is given in Algorithm 1. The weakness of the algorithm is the need of Q-values storage for every state-action pair. This could reason a huge amount of memory to be required for large dimensional real-world problems. Also, large state-action space could be required a huge number of episodes to find a successful policy. In this case, a generalization method is needed to suit a successful policy for the problem.

2.2 General Regression Neural Network

GRNN was proposed by Specht [25] is a memory-based neural network which does not require an iterative training. It simply keeps all training data as a pattern and forecasts the value of an input data with help of them.

GRNN is a four-layered network. The first layer is the input layer containing a neuron for each input value. The second layer is the pattern layer that keeps all training data as a pattern by holding a neuron containing a vector of data for each training data. When a new input data is entered into GRNN, it is subtracted for each stored data in pattern neurons. Then squares of differences are summed and the resulting value is passed through the Gaussian activation function. The third layer which is called summation layer has two neurons named as numerator and denominator. Numerator calculates a dot product between outputs of the

pattern layer and a vector which contains the expected values of each training data. Denominator calculates the sum of the outputs of the pattern layer. Finally, the output layer calculates the quotient of numerator and denominator values and finds the estimated value. Structure of the network is shown in Fig. 1. Equations of GRNN are shown with Eq. 2 and 3.

$$D_i^2 = (x - x_i)^{\mathrm{T}}(x - x_i), \tag{2}$$

$$Y(x) = \frac{\sum_{i=1}^{n} Y_i e^{\frac{-D_i^2}{2\sigma^2}}}{\sum_{i=1}^{n} e^{\frac{-D_i^2}{2\sigma^2}}}, \tag{3}$$

where x is the input vector, x_i is the data vector kept in the i-th pattern layer neuron, D_i is the distance between these two vectors. Y_i is the expected value of the i-th pattern layer neuron's data and $Y(s)$ is the expected value for the input vector given to the network.

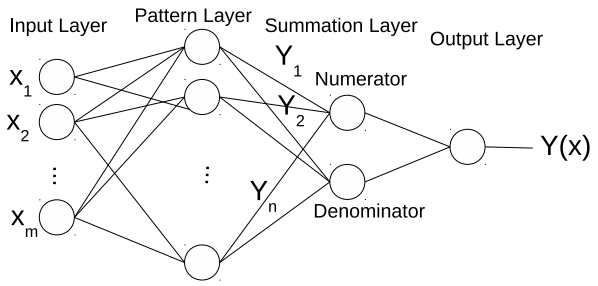

Fig. 1 *GRNN Structure.*

3. Q Learning Regression Neural Network

QLRNN is a generalized regression neural network for reinforcement learning problems. It is designed to be used for generalizing Q value function with the instance based working without an extra training phase. Other kernel-based regression methods use a predefined learning algorithm to be able to converge the kernel function parameters to be able to generalize the value function with it. On the contrary QLRNN only needs as accurate as possible data in the pattern layer to be able to predict the expected return of a state-action pair successfully. Firstly, an agent is executed for a number episodes with standard Q-learning algorithm to collect learning data. While the Q-Learning algorithm is producing data, QLRNN is established with the selected state-action pairs and target values of them as shown in Eq. 4.

$$\text{target} = r + \gamma \max_{a_{t+1}} Q(s_{t+1}, a_{t+1}), \tag{4}$$

where r is reward of the current state, γ is discount factor, $\max_{a_{t+1}} Q(s_{t+1}, a_{t+1})$ is the value of next state's best action. To limit the pattern layer size of the network, state-action space is divided into a finite number of pieces and then each piece of the

space is represented by a pattern neuron. The Q agent which collects training data also uses the same neurons as a look-up table. Only recent Q values of the pairs are also stored in an array. There is no need for any extra storage for the values of the standard Q learning algorithm. After each step of the training episode, observed triples (s, a, target) are replaced on the pattern layer of the network. This process is shown in Fig. 2. Agent decides its action with a predefined policy independent from the network and it runs the selected action on the environment. QLRNN algorithm can be called as an off-policy method for this reason.

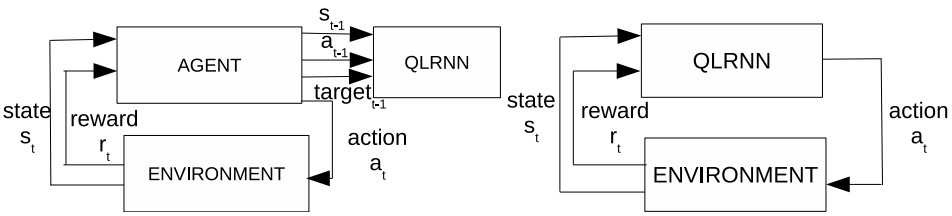

Fig. 2 *Training(left) and test(right) block diagrams of QLRNN.*

After a bunch of episodes is experienced, Q agent is run on the same environment with the QLRNN network for a number of different variance values to test the efficiency of the network. Current state parameters are passed through the QLRNN and action with the highest return expectation is selected for each step. This process is shown in Fig. 2.

3.1 QLRNN topology

In Fig. 3 QLRNN structure which is the same with that of GRNN is shown. All state and action parameters are input parameters of the network. State parame-

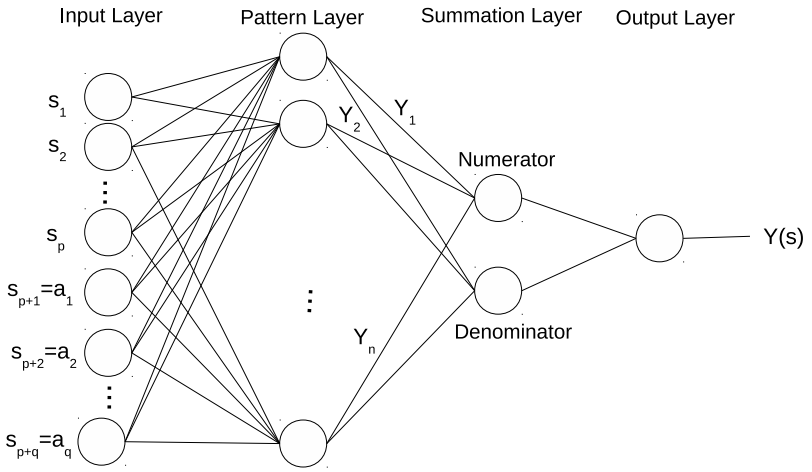

Fig. 3 *QLRNN Structure.*

ter(s) is/are p dimensional and action parameter/s is/are q dimensional. a action parameter/s is/are shown as state parameters on the figure. Calculation through network for n dimensional state space are given in Eq. 5 and 6 to simplify the projection.

$$D_i^2 = (s - s_i)^{\mathrm{T}}(s - s_i), \tag{5}$$

$$Y(s) = \frac{\sum_{i=1}^{n} Y_i e^{\frac{-D_i^2}{2\sigma^2}}}{\sum_{i=1}^{n} e^{\frac{-D_i^2}{2\sigma^2}}}, \tag{6}$$

where s is a vector that keeps all features of a state and action. σ is the bandwidth of the kernels. The output of the neural network is the reward expectation for action a on state s. D_i^2 is the square value of the weighted differences between the current state and states kept in the pattern layer. Y_i is the target value of the ith pattern layer node. $Y(s)$ is the expected return of the given state-action pair.

QLRNN is a four-layered network. Input layer includes a neuron for each state and action parameter. Pattern layer keeps all training data as a pattern. It contains a neuron containing state action values for each training data. When a state action pair is entered into QLRNN, it is subtracted from each stored pair in pattern neurons. After squares of differences are collected, the resulting value is passed through the Gaussian activation function. Summation layer has two neurons named as numerator and denominator. Numerator calculates a dot product between outputs of the pattern layer and a vector which contains expected return values for each training state-action pair. Denominator calculates the sum of the outputs of the pattern layer. Finally, the output layer calculates the quotient of numerator and denominator values and finds the estimated return value.

3.2 QLRNN learning algorithm

In QLRNN algorithm, the pattern layer is empty before the training phase and the maximum number of nodes in the pattern layer must be decided by dividing the state-action space into a finite number of pieces. Each piece will be represented by a neuron on the pattern layer. If pieces are small-sized, the number of neurons will increase and the pace of the QLRNN will slow down. On the other hand, if pieces are large-sized, the number of neurons will decrease, the pace of the QLRNN will rise up, however, this time QLRNN may not be able to converge a good solution. Number of the pattern layer neurons must be decided carefully due to this fact. After the decision of neuron numbers, Q agent uses a predefined policy such as ε-greedy. This policy proposes selecting the best action with $1 - \varepsilon$ possibility or to select a random action with ε possibility where $0 < \varepsilon \leq 1$. Q-agent is run on the environment for a decided number of episodes. While these episodes are run, lookup table values are updated with equation given in Eq. 1 and target values are calculated according to Eq. 4. Standard Q agent generating training data uses the pattern layer as a lookup table for state-action values. If state action pair is chosen for the first time value of the pair will be taken zero. After each step of an episode a neuron representing an unexisting state is added or existing representing neuron is updated according to recent values of the state action pair. In other words, the relevant neuron is added or updated on the pattern layer for each step of the

episode. A pattern layer neuron keeps a tuple containing state, action and target values. Calculation of target value is shown in Eq. 4. After a number of episodes, QLRNN efficiency is measured on the problem directly. Agent starts from an initial state. Reward expectation is calculated through QLRNN according to Eq. 5 and 6 for each possible action. The action owns the highest return expectation is chosen. This process is repeated till episode is completed. The test is done with a sufficient number of episodes. If QLRNN is efficient enough according to the measuring criteria, all process will be completed otherwise Q agent will run on the environment for another number of episodes to update the QLRNN. The whole process is shown in Algorithm 2.

Algorithm 2 QLRNN algorithm.

Set all $Q(s,a) = 0$
Repeat forever:
Repeat (for n episode):
Set s as one of the initial states
Repeat (for each step of the episode)
Select action a according to s using policy derived from Q
Take action a, observe r, next state s_{t+1}
$Q(s,a) = Q(s,a) + \alpha[r + \gamma \max_{a_{t+1}} Q(s_{t+1}, a_{t+1}) - Q(s,a)]$
target $= r + \gamma$
$\max_{a_{t+1}} Q(s_{t+1}, a_{t+1})$
If (s,a) is exist in the pattern layer of QLRNN
update pattern neuron with (s,a) and target values
else
add a new neuron to QLRNN with (s,a) and target values
$s \leftarrow s_{t+1}$
until s is terminal
Measure QLRNN efficiency with k random episodes
If QLRNN is efficient break

Normalization process can be needed due to the range of the state and action values. If normalization is done pattern layer neurons will keep the normalized values. All state and action values must also be normalized before the value passes over the QLRNN for estimating process. In this paper, min-max normalization is implemented for all benchmarks.

In the Q learning algorithm, the Q function approaches optimality with each iteration. Q_{t+1} will generate more accurate values than Q_t. Convergence of Q learning is showed with a method called ARP(action-replay process) in the original article [2]. Since QLRNN is a representation of the current Q function, $g(Q)$, it will also approach optimality over time. $g(Q_{t+1})$ will have a better solution than $g(Q_t)$) for the problem to be solved. In addition, the QLRNN output function is defined within the entire space, even with a single neuron as it seen in Eq. 6. Thus, for all possible input values, output value is defined, and QLRNN produces a valid value at any point in the space.

The outermost loop used for testing purposes can be ignored, so the complexity of the algorithm is computed as $O(n^2)$. The space complexity of the algorithm

grows to be equivalent to the space complexity of the problem to be solved. To give an example, three different values are needed to determine a point in a 3D world. In a QLRNN to be used for such an environment, each neuron will hold 3 different values for a point.

4. Test results and discussion

Three different popular benchmarks were used for measuring the quality of QL-RNN generalization. These benchmarks are the pole-balancing task, mountain-car benchmark, and puddle-world which are commonly used for measuring the performance of reinforcement learning algorithms. All training and testing operations were done on Closed Loop Simulation System which was supported by Riedmiller [26] and also used for testing NFQ algorithm.

4.1 The pole-balancing task

In the two-dimensional world, a cart with a pole onto is placed on a track. There exist three actions with different forces ($-50\,\mathrm{N}$, $0\,\mathrm{N}$, $+50\,\mathrm{N}$). The goal is to prevent the pole to fall. There is only one negative reward (-1) which is gained when the pole falls. There are two parameters associated with the problem: θ is the position of the pole in radian, θ_v is the angular velocity of the pole. Value range of the parameters are $\theta\epsilon[-\frac{\pi}{2}, \frac{\pi}{2}]$ rad and $\theta_v\epsilon[-2.5, 2.5]\,\mathrm{rad/s}$ for θ and θ_v respectively.

Transitions between states are determined by the nonlinear dynamics of the system [27] and each action changes acceleration of the pole according to Eq. 7

$$\theta_a(t) = \frac{g\sin\theta(t) - \cos\theta(t)\left(\frac{(F(t)+ml\theta_v(t)^2\sin\theta(t))}{m_c+m}\right)}{l\left(\frac{4}{3} - \frac{m\cos^2\theta(t)}{m_c+m}\right)}. \tag{7}$$

$g = 9.8\ \mathrm{m/s^2}$ is the gravity, $l = 0.5$ m is the half-pole length, $F(t) = F + \eta$ is the force where η is the noise term valued in the range of $[-10, 10]$. $m = 2\,\mathrm{Kg}$ and $m_c = 8\,\mathrm{Kg}$ are the masses of the pole and the cart respectively. Position and velocity the pole is changed by the Eqs. 8 and 9 after each time step which is $\tau = 0.1\,\mathrm{s}$. Benchmark parameters are completely the same as the benchmark used for measuring NFQ and LSTD regression performances

$$\theta(t) = \theta(t) + \tau\theta_v(t), \tag{8}$$

$$\theta_v(t) = \theta_v(t) + \tau\theta_a(t). \tag{9}$$

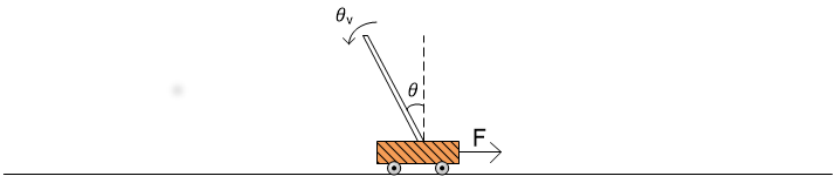

Fig. 4 *Pole-balance (inverted-pendulum) benchmark.*

Results

θ and θ_v were used as state parameters. State-action space was divided into $(16 \cdot 10 \cdot 3)$ pieces. 16 different interval for θ, in case each neuron represents a range amount of $\frac{\pi}{16}$ for θ on the space. 10 different interval for θ_v, in case each neuron represents a range amount of 0.5 for θ_v on the space. The last 3 different pieces represent the 3 discrete actions. In this case, probable maximum number of pattern nodes become 480. There are unreasonable states which cannot be visited, in case where the number of nodes in the pattern layer was not above 200. Training episodes start with $\theta = 0$ and $\theta_v = 0$ initial values. The maximum step size of a training episode was limited to 3000. All parameters were normalized with min-max normalization with a range of $[-11]$. Each movement represents a 0.1 second of motion in real time which results in 300 seconds (5 minutes) of overall duration. After each training episodes, QLRNN performance was measured with different variance values between 0.002 and 0.3 for different 1000 test episodes on the benchmark. $\alpha = 0.5$ and $\gamma = 1$ was set for the Q agent. ε-greedy exploration was used with $\varepsilon = 0.2$.

25 different experiments have been done. It was obtained that QLRNN needs averagely 23 episodes to learn a successful policy that keeps the pole balanced for more than 300 seconds. Mean and standard deviation of successful results are shown in Tab. I. It was presented by Lagoudakis and Parr [15], LSPI can balance the pole about 285 seconds after 1000 episodes. Q learning with experience replay, suggested in the same paper, requires averagely 700 episodes to learn a successful policy. It was reported by Riedmiller NFQ algorithm needs averagely 200 episodes to find a successful policy [17]. However, each episode was repeated over the network 50 times to train with backpropagation. In this case, training of the network takes 10000 passes. Since QLRNN is not a type of backpropagation network. After averagely 23 training episodes are run, neuron organization of the neural network is done and ready to use. The comparison between the performance of algorithms is shown in Tab. II.

Training Lenght for Successful Policy			
Episodes		Cycles	
Mean	Std. Dev.	Mean	Std. Dev.
22.840	7.284	289.040	128.723

Tab. I *Pole-balancing task results.*

4.2 Mountain car benchmark

The mountain car problem was firstly defined by Moore [28]. When Singh and Sutton added the problem into "Reinforcement Learning: An Introduction" book [1], it became more popular. The problem is about a powerless car must overcome a hill. Since the gravity is stronger than the car's engine; the car can not speed up and reach the peak. The car is settled on a valley and must learn to keep potential energy by driving up the opposite hill. After that, the car becomes able to reach the peak of the rightmost hill.

	Training Length		Pass over network	Balancing Time
	Episodes	Cycles		[s]
LSPI	1000.00	–	–	285.00
LSPI with ER	700.00	–	–	300.00
NFQ	200.00	1200.00	50.00 times	300.00
QLRNN	22.84	289.04	no need	300.00

Tab. II *Pole-balancing task results.*

There are two parameters associated with the problem: V is the velocity of the car and P is the position of the car. Value range of the parameters are $V\epsilon(-2.5, +2.5)$ and $P\epsilon(-1.0, 0.7)$ for V and P respectively. There exist two different actions -4 and $+4$ which change position and velocity of the car according to Eqs. 10 and 11. All parameters in the equations are kept the same with NFQ test benchmark. In the case of a failure, the episode 5is terminated and rewarded by -1. This occurs when the position of the car calculated less than -1.0. Otherwise, each step is rewarded by -1 unless reaching the goal state

$$V = V + 0.001 \cdot \text{Action} + \cos(3P) \cdot (-0.0025), \tag{10}$$

$$P = P + V. \tag{11}$$

At the initial condition, a position is chosen randomly and velocity is set to 0. When the position becomes higher than 0.7, the episode is terminated with a reward of 0.

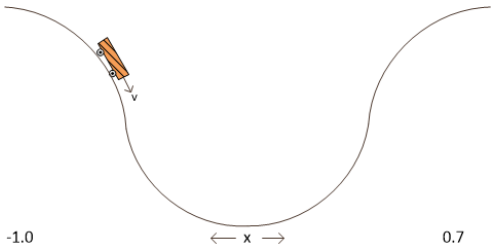

Fig. 5 *Mountain-Car Benchmark.*

Results

State-action space was defined in the size of $(17 \cdot 10 \cdot 2)$. 17 different interval for the position parameter and 10 different intervals for the velocity parameter were determined. The maximum number of pattern nodes become 396 for 2 discrete actions. Some of the nodes were not created because of unvisited or irrational state-action pairs during the episodes. Therefore the number of nodes in the pattern layer became at most 300. Training episodes start with a random position and zero velocity. The maximum step size of a training episode was limited to 50. After each episode, QLRNN performance was measured with different variance values

between 0.001 and 0.5 for random 1000 initial states on the benchmark. $\alpha = 0.5$ and $\gamma = 1$ was set for the Q agent. ε-greedy exploration was chosen as $\varepsilon = 0.2$. All parameters were normalized with min-max normalization with a range of $[-11]$.

QLRNN was trained and performance of QLRNN was measured with given parameters for 25 times. Test episodes were limited with the maximum number of 300 steps. 500 training episode was used for each experiment. All successful policies were determined. First and best Q-agent policies are shown in Tab. III.

First Successful Policy					
Episodes		Cycles		Ave. Cost	
Mean	St.D.	Mean	St.D.	Mean	St.D.
31.28	27.53	1073.00	957.40	41.66	15.32
Best Successful Policy					
Episodes		Cycles		Ave. Cost	
Mean	St.D.	Mean	St.D.	Mean	St.D.
171.24	162.53	5960.00	5561.00	29.18	3.09

Tab. III *Mountain car benchmark results.*

After a sufficient amount of episodes, it can be seen that average cost values approach to the optimum. QLRNN was approximately 2 times faster than NFQ for Mountain-Car benchmark with nearly the same average cost values as shown in Tab. IV.

	First Succesful Policy			Best Succesful Policy		
	Episodes	Cycles	Average Cost	Episodes	Cycles	Average Cost
NFQ	70.95	2777.00	41.05	296.60	10922.80	28.70
QLRNN	31.28	1073.16	41.66	171.24	5960.48	29.18

Tab. IV *Mountain car benchmark results.*

The Same benchmark was also run with the same parameters given by Dutech et al. [19]. In this experiment 3 different actions $(-1, 0, 1)$ and position range between $(-1.1, 0.5)$ were used. QLRNN was run for 500 training episodes. In this case, QLRNN was able to find a successful policy with 53.05 episodes with a cost of 111.21 averagely. Best policy found by QLRNN had an average cost of 85.47. LEAP is not able to decrease the average cost under 100 within 10000 training episodes. RMAX is able to find a policy with an average cost of 85.59 after more than 1000 training episodes.

Mountain-car benchmark was also run with the same parameters given by Whiteson and Stone [20]. This time 3 different actions $(-1, 0, 1)$ and position range between $(-1.2, 0.5)$ were used. QLRNN was able to find a successful policy with an average of 48.35 episode with an average cost of 116.11. Best policy found by QLRNN had an average cost of 81.24 with 5.52 standard derivation which was near to the optimal cost. NEAT-Q algorithm converges slower and not able to decrease the cost under 100 in the cited paper. LSPI is also able to find nearly the optimal

policy, however, LSPI averagely needs 200 episodes to generate a policy with a cost of less than 100 while QLRNN needs averagely just 118 episodes. QLRNN was also able to find an appropriate solution faster than XAI. Comparison of the three algorithms can be seen in Fig. 6. Cost values of QLRNN were calculated by averaging cost values of the best policy found till the end of the episodes.

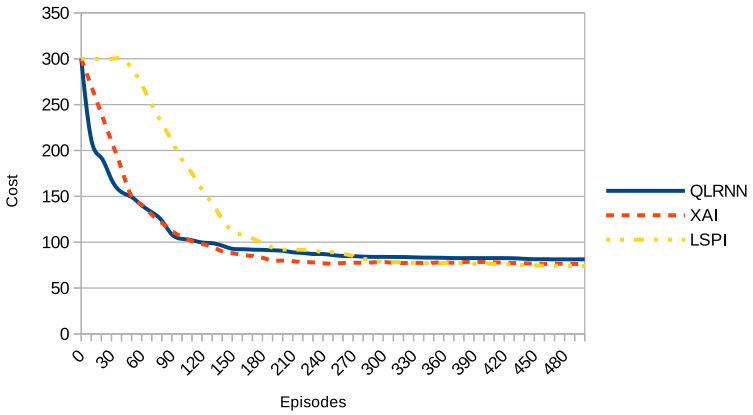

Fig. 6 *QLRNN performance on mountain car.*

4.3 Puddle-world problem

This problem was defined by Sutton [3]. The aim of the task is to move a robot in a continuous 2D area to a goal which is unknown by the robot with minimum cost. The area is defined as $[0, 1]^2$. There are two puddles located in the area as shown in Fig. 7. The axes indicate the position on the 2D world on the figure.

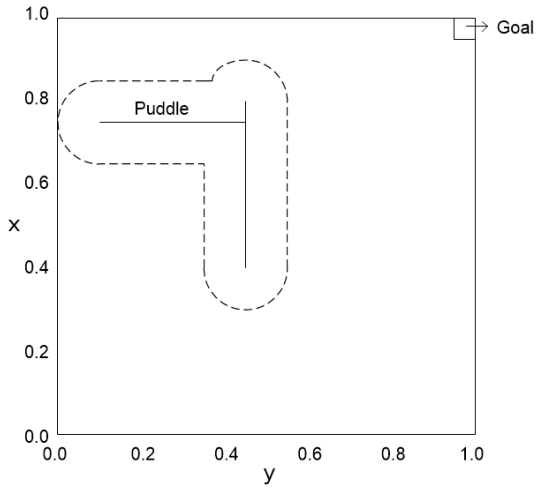

Fig. 7 *Puddle world benchmark.*

The puddles are oval shaped with a radius of 0.1 and they are stated at centerlines $(0.1, 0.75)$ to $(0.45, 0.75)$ and $(0.45, 0.4)$ to $(0.45, 0.8)$. There are two state variables and four discrete actions for each direction. Each action moves the robot by 0.05 to its direction. A random gaussian noise up to 0.01 is added to each direction on each move. Starting position is chosen randomly in the area. The goal areas are defined with $x + y \geq 0.95 + 0.95 = 1.9$. Each time step is costed by -1. Actions taken inside the puddles have an extra cost which is calculated by multiplying the distance inside the puddle by -400.

Results

State-action space was defined in the size of $10 \cdot 10 \cdot 4$. The maximum number of pattern nodes became 400 for 4 discrete actions. Training episodes start with a random position over the 2D area. The maximum step size of a training episode is limited to 500. After each episode QLRNN performance is measured with different variance values between 0.001 and 0.5 for random 1000 initial states on the benchmark. α and γ were set to 0.5 and 1 for the Q agent. ε-greedy exploration was used with $\varepsilon = 0.2$. All parameters were normalized with min-max normalization with a range of $[-1, 1]$.

QLRNN was trained and performance of QLRNN was measured with given parameters for 20 times. Test episodes were limited with a maximum number of 500 steps. 500 training episode was used for each experiment. All successful policies were determined.

First Succesful Policy				Cost$\leq$ 40		Cost$\leq$ 30		Best	
Episodes		Ave. Cost		Episodes		Episodes		Ave. Cost	
Mean	St.D.	Mean	St.D.	Mean	St.D.	Mean	St.D.	Mean	St.D.
154.75	44.05	42.98	15.48	179.75	60.31	203.50	81.64	22.86	0.55

Tab. V *Puddle world results.*

QLRNN was able to find a successful policy which can lead the robot to the goal with an average of 154.75 episodes with an average cost of 42.98. Average best cost result of all trials was 22.86 which is nearly the optimal cost for the benchmark. LEAP algorithm needs more than 1000 episodes to get the average cost under 50. LSPI converges a solution with an average cost of 35.54 after 500 episodes. RMAX can also able to find the optimal solution with an average cost of 22.85 but after 1000 training episodes averagely. XAI converges a solution with an average cost of 30 after more than 1000 training episodes. QLRNN can find a quick appropriate solution and it is also able to find the optimal solution faster than other algorithms for the puddle-world benchmark.

5. Conclusion

In this paper, a Nadaraya-Watson kernelled regression method with GRNN topology is adapted to Q learning algorithm to generalize the value function and to generate a successful action selection policy. Along the methodology, a roadmap is

also given about how to organize the pattern layer of the instance based working network. The high efficiency of regression is obtained by the utilization of GRNN inherited network topology. On the other hand, the main drawbacks of GRNN are also contained by QLRNN. These are mainly problematic of the selection of the efficient variance value for the corresponding data set and relatively high throughput time for a new data due to the complex calculation. Estimated time consumption is decreased by limiting the number of pattern layer nodes of the QLRNN.

Under a limited number of pattern layer neurons, QLRNN is also effective with real-time learning. It is formed while Q agent is running on the environment with a predetermined policy. After that its performance is tested on the same problem directly. QLRNN does not require an extra training time due to its instance based structure. In error-minimizing networks, a bunch of episodes must be stored as training data and then they must be passed through the network for many epochs. If the learning process is unsuccessful, a new or bigger data set must be tried for the network. In contrast, QLRNN evolution process can be done by growing the states kept in pattern layer. After a number of episodes used for establishing QLRNN, additional episodes can also be used for improving the performance of the network. Through this work, QLRNN is applied to solve popular problems of reinforcement learning that are mountain car, pole balance, and puddle-world problems. Regression performance of QLRNN is compared with that of LSPI, NFQ, XAI, LEAP, RMAX, NEAT-Q algorithms on the test benches. QLRNN learns faster than other algorithms on selected problems. Comparisons are done by considering the number of learning steps. In mountain car problem the learning is accelerated more than 2 times against NFQ algorithm. QLRNN learns much faster and more efficient than LEAP, NEAT-Q, and RMAX on the same problem. While LSPI and XAI are able to find better policies over the long run, QLRNN is able to find an appropriate policy much faster than them on the mountain-car benchmark. In pole balance problem QLRNN is faster approximately 30 times than LSPI and it is also faster approximately 4 times than NFQ. In the puddle-world benchmark, QLRNN's performance and efficiency are much better than all compared algorithms. These test results prove the efficiency and show the importance of the proposed network.

References

[1] SUTTON R.S., BARTO A.G. *Introduction to reinforcement learning*. MIT Press, 1998 doi: 10.1007/978-3-319-20010-1_14.

[2] WATKINS C.J.C.H., DAYAN P. Q-learning *Machine learning*. 1992, 8(3-4), pp. 279–292, doi: 10.1007/springerreference_57860.

[3] SUTTON R.S. Generalization in reinforcement learning: Successful examples using sparse coarse coding *Advances in neural information processing systems*. 1996, pp. 1038–1044.

[4] TIAN Z., LI S., WANG Y., WANG X. A network traffic hybrid prediction model optimized by improved harmony search algorithm. *Neural Network World*, 2015, 25(6), pp. 669. doi: 10.14311/NNW.2015.25.034

[5] KALAIE A., SHUJIANG L., YANHONG W., XIANGDONG W. Network traffic prediction method based on improved ABC algorithm optimized EM-ELM. *The Journal of China Universities of Posts and Telecommunications*, 2018, 25(3), pp. 33–44, doi: 10.19682/j.cnki.1005-8885.2018.0014.

[6] TIAN Z., LI S., WANG Y. TS fuzzy neural network predictive control for burning zone temperature in rotary kiln with improved hierarchical genetic algorithm. *International Journal*

of Modelling, Identification and Control, 2016, 25.4, pp. 323-334, doi: 10.1504/IJMIC.2016.076825.

[7] TIAN Z., WANG G., REN Y., LI S., WANG Y. An adaptive online sequential extreme learning machine for short-term wind speed prediction based on improved artificial bee colony algorithm. *Neural Network World*, 2018, 28.3, pp. 191-212, doi: 10.14311/NNW.2018.28.012.

[8] ZHONGDA T., SHUJIANG L., YANHONG W., YI S. A prediction method based on wavelet transform and multiple models fusion for chaotic time series. *Chaos, Solitons & Fractals*, 2017, 98: pp. 158-172, doi: 10.1016/j.chaos.2017.03.018.

[9] TIAN Z., LI S., WANG Y. The Multi-Objective Optimization Model of Flue Aimed Temperature of Coke Oven. *Journal of Chemical Engineering of Japan*, 2018, 51.8: pp. 683-694, doi: 10.1252/jcej.17we159.

[10] ANDERSON C.W. Strategy learning with multilayer connectionist representations *Proceedings of the Fourth International Workshop on Machine Learning*,1987, pp. 103–114, doi: 10.1016/b978-0-934613-41-5.50014-3.

[11] BOYAN J.A., MOORE A.W. Generalization in reinforcement learning: Safely approximating the value function *Advances in neural information processing systems*. 1995, pp. 369–376.

[12] BRADTKE S.J., BARTO A.G. Linear least-squares algorithms for temporal difference learning *Machine learning*. 1996, 22(1-3), pp. 33–57, doi: 10.1007/978-0-585-33656-5_4.

[13] BOYAN J.A. Technical update: Least-squares temporal difference learning *Machine Learning*. 2002, 49(2-3), pp. 233–246, doi: 10.1007/978-0-585-33656-5_4.

[14] BRAFMAN R.I., TENNENHOLTZ M. R-max-a general polynomial time algorithm for near-optimal reinforcement learning *Journal of Machine Learning Research*. 2002, 3(Oct), pp. 213–231.

[15] LAGOUDAKIS M.G., PARR R. Least-squares policy iteration *The Journal of Machine Learning Research*. 2003, 4, pp. 1107–1149.

[16] ERNST D., GEURTS P., WEHENKEL L. Tree-based batch mode reinforcement learning *Journal of Machine Learning Research*, 2005, pp. 503–556.

[17] RIEDMILLER M. Neural fitted Q iteration–first experiences with a data efficient neural reinforcement learning method *Machine Learning: ECML 2005*, 2005, pp. 317–328, doi: 10.1007/11564096_32.

[18] BONARINI A., LAZARIC A., RESTELLI M. Reinforcement learning in complex environments through multiple adaptive partitions *Congress of the Italian Association for Artificial Intelligence*, 2007, pp. 531–542, doi: 10.1007/978-3-540-74782-6_46.

[19] DUTECH A., EDMUNDS T., KOK J., LAGOUDAKIS M., LITTMAN M., RIEDMILLER M., RUSSELL B., SCHERRER B., SUTTON R., TIMMER S., others. Reinforcement learning benchmarks and bake-offs II *Advances in Neural Information Processing Systems (NIPS)*. 2005, 17.

[20] WHITESON S., STONE P. Evolutionary function approximation for reinforcement learning *The Journal of Machine Learning Research*, 2006,7, pp. 877–917, doi: 10.1007/978-3-642-13932-1_4.

[21] HEINEN M.R., ENGEL P.M. IPNN: An incremental probabilistic neural network for function approximation and regression tasks *Neural Networks (SBRN), 2010 Eleventh Brazilian Symposium*. 2010, pp. 25–30, doi: 10.1109/sbrn.2010.13.

[22] HEINEN M.R., ENGEL P.M. An incremental probabilistic neural network for regression and reinforcement learning tasks *Artificial Neural Networks–ICANN 2010*, 2010, pp. 170–179, doi: 10.1007/978-3-642-15822-3_22.

[23] LAMPLE G., CHAPLOT D.S. Playing FPS Games with Deep Reinforcement Learning. In: AAAI, 2017, pp. 2140-2146.

[24] SROUJI M., ZHANG J., SALAKHUTDINOV R. Structured Control Nets for Deep Reinforcement Learning. arXiv preprint arXiv:1802.08311, 2018.

[25] SPECHT D.F. A general regression neural network *Neural Networks, IEEE Transactions on*, 1991,2(6), pp. 568–576, doi: 10.1016/s0893-6080(09)80013-0.

[26] Closed Loop Simulation System 2013 [accessed 2016-12-06]. Available from: http://ml.informatik.uni-freiburg.de/research/clsquare

[27] WANG H.O., TANAKA K.G., MICHAEL F. An approach to fuzzy control of nonlinear systems: stability and design issues *Fuzzy Systems, IEEE Transactions on*, 1996, 4(1), pp. 14–23, doi: 10.1109/91.481841.

[28] MOORE A.W., HALL T. Efficient memory-based learning for robot control, 1990.