ÇUKUROVA UNIVERSITY
INSTITUTE OF NATURAL AND APPLIED SCIENCES

MSc THESIS

Mehmet SARIGÜL

Q REGRESSION NEURAL NETWORK

DEPARTMENT OF COMPUTER ENGINEERING

ADANA, 2014

ÇUKUROVA UNIVERSITY
INSTITUTE OF NATURAL AND APPLIED SCIENCES

Q REGRESSION NEURAL NETWORK

Mehmet SARIGÜL

MSc THESIS

DEPARTMENT OF COMPUTER ENGINEERING

We certify that the thesis titled above was reviewed and approved for the award of degree of the Master of Science by the board of jury on ../../2015.

……	……...	……..
Assoc. Prof. Dr. Mutlu AVCI	Assoc. Prof. Dr. Selma A. Özel	Asst. Prof. Dr. Serdar YILDIRIM
SUPERVISOR	MEMBER	MEMBER

This MSc Thesis is written at the Department of Institute of Natural And Applied Sciences of Çukurova University.
Registration Number:

Prof. Dr. Mustafa GÖK
Director
Institute of Natural and Applied Sciences

This thesis is supported by TÜBİTAK.

Not: The usage of the presented specific declerations, tables, figures, and photographs either in this thesis or in any other reference without citiation is subject to "The law of Arts and Intellectual Products" number of 5846 of Turkish Republic

ABSTRACT

MSc THESIS

Q REGRESSION NEURAL NETWORK

Mehmet SARIGÜL

ÇUKUROVA UNIVERSITY
INSTITUTE OF NATURAL AND APPLIED SCIENCES
DEPARTMENT OF COMPUTER ENGINEERING

Supervisor : Assoc. Prof. Dr. Mutlu AVCI
Year: 2015, Pages: 43
Jury : Assoc. Prof. Dr. Mutlu AVCI
: Assoc. Prof. Dr. Selma Ayşe ÖZEL
: Asst. Prof. Dr. Serdar YILDIRIM

Q Learning was an important novelty for reinforcement learning. However Q learning becomes inapplicable due to the gigantic size of solution spaces of real world problems. Therefore, an efficient regression method is required not only to generalize the state, action space but also accelerate speed of the learning algorithm.

In this thesis, Q regression neural network suggested is a novel regression method obtained as a result of a GRNN form structure used for generalization Q-value function. QRNN works with data generated by Q-Agent. It is tested with popular reinforcement learning benchmarks and its performance is compared with that of both Q-Learning and the other regression methods. Test results show that, QRNN learning efficiency is much higher than the compared methods.

Key Words: Reinforcement Learning, Q Learning, Q value function generalization, GRNN

ÖZ

YÜKSEK LİSANS TEZİ

Q REGRESYON SİNİR AĞI

Mehmet SARIGÜL

ÇUKUROVA ÜNİVERSİTESİ
FEN BİLİMLERİ ENSTİTÜSÜ
BİLGİSAYAR MÜHENDİSLİĞİ BÖLÜMÜ

Danışman : Doç. Dr. Mutlu AVCI
Yıl: 2015, Sayfa: 43
Jüri : Doç. Dr. Mutlu AVCI
: Doç. Dr. Selma Ayşe ÖZEL
: Yrd. Doç. Dr. Serdar YILDIRIM

Q öğrenme yöntemi takviyeli öğrenme için önemli bir gelişme olmuştur. Ancak gerçek dünya problemlerinin sahip olduğu devasa büyüklükteki durum-aksiyon sayısı Q öğrenme yöntemini uygulanamaz hale getirmektedir. Bu durum sadece durum-aksiyon uzayını genellemeyecek, aynı zamanda da öğrenmeyi hızlandıracak etkili bir regresyon metodunu gerekmektedir.

Bu tezde Q regresyon yapay sinir ağı adında Q değer fonksiyonunun genelleştirilmesi ile GRNN formunda bir yapı kullanılması sonucu elde edilen yeni bir eğiticisiz yapay sinir ağı önerilmiştir. QRNN Q ajanının ürettiği veriler üzerinde çalışır. Önerilen metot popüler takviyeli öğrenme çalışma ortamları üzerinde test edilmiş, Q öğrenme yöntemi ve diğer regresyon yöntemleri ile karşılaştırılmıştır. Test sonucunda öğrenme hızının karşılaştırılan yöntemlere göre oldukça yüksek olduğu gözlenmiştir.

Anahtar Kelimeler: Takviyeli Öğrenme, Q öğrenme yöntemi, Q değer fonksiyonu genelleme, GRNN

ACKNOWLEDGEMENTS

I would like to thank my advisor Assoc. Prof. Dr. Mutlu AVCI for his continuous and unconditional support, valuable guidance and encouragement.

I would like to thank each and every member of the evaluation committee for their guidance.

I also want to thank my parents and my wife who never stopped believing in me. I appreciate all of their support and encouragements.

I would like to thank TÜBİTAK for financial support during my MSc education.

CONTENTS **PAGE**

LIST OF TABLES **PAGE**

LIST OF FIGURES **PAGE**

1. INTRODUCTION

Machine learning, a branch of artificial intelligence, concerns the construction and study of algorithms and systems that can learn from data. When it is thought about nature of learning, it can be realized how the idea of learning by interacting with environment is emerged. When a car is driven or a speech is given, driver or presenter is aware of how environment will respond to the action which is done. Reinforcement learning is a computational learning approach that based on learning from interacting with the environment.

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 total amount of rewards as a result of the selected actions. Q-Learning is a one of the popular and successful reinforcement learning methods. All state-action pairs which are existing in environment are valued depend to rewards that they give and values of next state-action pairs that they bring into.

In this thesis, proposed QRNN is a new artificial neural network containing Q-learning and general regression approach simultaneously. It may be considered as a reinforcement learning general regression neural network with Q-learning. It works with approximate values of state-action pairs. After obtaining target values according to existing experience of Q-agent, these values are used as expected return values of the neural network. To measure the efficiency of the network, QRNN is used directly on problems which Q-agent may work on them. QRNN is not only generalizing Q-values, it also increases rate of successful action selections. Finally, after tests it is observed that QRNN learns faster than standard Q-Learning algorithm and also other popular regression methods such as LSPI (Least Squares Policy Iteration) and NFQ (Neural Fitted Q Iteration). It is also more efficient than the mentioned algorithms. Tests are done by considering number of learning steps. In mountain car problem the learning is accelerated more than 100 times. In pole balance problem with two parameters; QRNN is faster approximately 1.4 times than LSPI and it is also faster approximately 19 times than NFQ. In pole balance problem with four parameters,

QRNN is approximately 1.11 times faster than NFQ although QRNN is implemented in a wider state action space. These test results prove the efficiency and show importance of the proposed network.

1.1. Reinforcement Learning

Reinforcement learning can be identified by four main sub elements: policy, reward function, value function, and model of the environment. A policy defines agent's way of behaving at a given situation. Reward function determines the amount of the award that is gained by the agent as a result of the chosen actions. Value function determines how profitable being in a state for future actions. A high-quality state indicates the expectation of winning high-valued prizes by moving to the state. Model is the last sub element is all of the environmental factors (Watkins 1989).

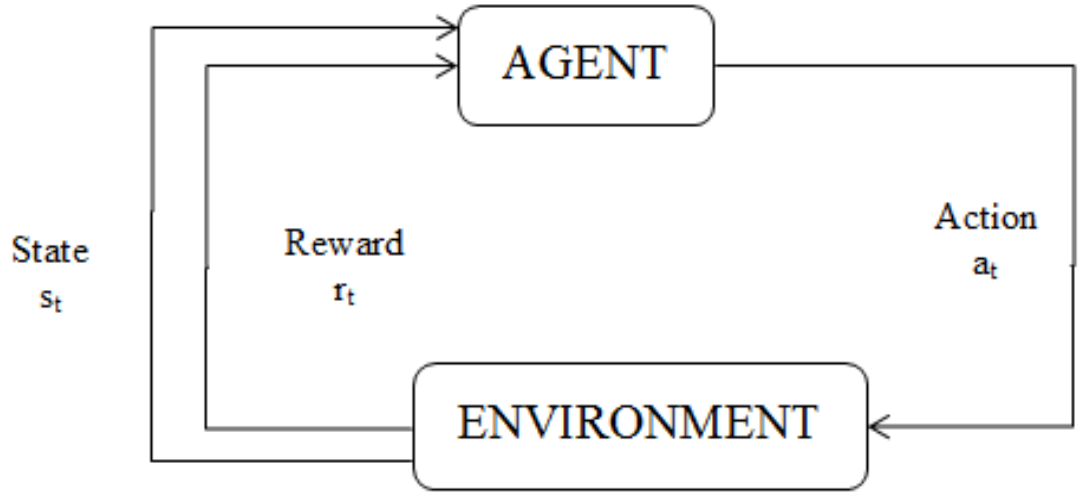


Figure 1.1. Agent Environment Communication

The target of the reinforcement learning is to find an optimal policy that will be maximizing the rewards to be won over the long run.

All reinforcement learning algorithms are based on the *"value function"* that specifies how valuable is being in a state and how good to be in a state for an agent (Bellman 1957).

$$V^{\pi}(s) = \sum_{a} \pi(s,a) \sum_{s'} P^{a}_{ss'} [R^{a}_{ss'} + \gamma V^{\pi}(s')] \quad (1.1)$$

$V^{\pi}(s)$ indicates value of state s. $\pi(s,a)$ is the probability of occurence of the action a in state s under the policy π . $P^{a}_{ss'}$ is the probability of transition to state s' after action a. $R^{a}_{ss'}$ is the reward that is gained with the action a on transition from s to s'. $V^{\pi}(s')$ is the value of the new state s'. γ is a discount rate parameter that specifies how much would be the effect of the next state value on the updating of the previous state's value. Solving a reinforcement learning problem means finding an optimal policy that specifies how to act against different states. *Bellman optimal value function* can be used to find an optimal policy (Bellman 1957).

$$V^{*}(s) = \max_{\pi} V^{\pi}(s) \quad (1.2)$$

1.1.1. Dynamic Programming

Dynamic programming methods use Bellman equations iteratively to calculate exact values of the states. A great model of all environmental factors is needed to be able to use dynamic programing. It is hard to use for models that have lots of states due to the calculations must be done for all states.

Dynamic programming is used the value assignment function iteratively and state values are updated iteratively.

$$V_{k+1}(s) = \sum_{a} \pi(s,a) \sum_{s'} P^{a}_{ss'} \left[R^{a}_{ss'} + \gamma V_{k}(s')\right] \quad (1.3)$$

Last optimal policy is determined when changes in the values of states do not cause a change in the current optimal policy.

$$\pi'^{(s)} = \underset{a}{\operatorname{argmax}} \sum_{s'} P^{a}_{ss'} \left[R^{a}_{ss'} + \gamma V^{\pi}(s') \right] \tag{1.4}$$

Optimal policy defines a path that begins from the first state and follows the best actions that move the agent to the best states until the final state is arrived.

1. Initialization

V(s)ϵR and π(s)ϵA(s) arbitrarily for all sϵS

2. Policy Evaluation

Repeat:

Δ←0;

For each sϵS

v←V(s);

$$V(s) = \sum_{s'} P^{\pi(s)}_{ss'} \left[R^{\pi(s)}_{ss'} + \gamma V(s') \right]$$

$$\Delta \leftarrow \max(\Delta, |v - V(s)|)$$

Until Δ<θ (a small positive number)

3. Policy Improvement

policy_stable←true

For each sϵS:

b← π(s)

$$\pi(s) \leftarrow \underset{a}{\operatorname{argmax}} \sum_{s'} P^{a}_{ss'} \left[R^{a}_{ss'} + \gamma V(s') \right]$$

If b≠π(s) then policy_stable←false

If policy_stable=true then stop, else go to 2.

Figure 1.2. Dynamic Programming Algorithm

1.1.2. Temporal Difference Learning

TD methods advise to use experience to solve the prediction problem between actions. After taking each action, the value of the previous state is updated depending on the new state's value and gained reward (Sutton, 1988).

$$V(s_t) = V(s_t) + \propto [r_{t+1} + \gamma V(s_{t+1}) - V(s_t)] \quad (1.5)$$

$V(s_t)$ is value of the current state. $V(s_{t+1})$ is value of the following state. r_{t+1} is the reward obtained in transition from s_t to s_{t+1}. $\propto$ is a constant that specify how quickly the state values will change. γ is another constant that specifies how much would be the effect of the next state value on the previous state value on the update the process.

Initialize V(s) arbitrarily, π to the policy to be evaluated
Repeat (for each episode):
    Initialize s
    Repeat (for each step of episode)
        a← action given by π for s
        Take action a; observe reward, r, and next state, s'
        $V(s) \leftarrow V(s) + \alpha[r + \gamma(V(s') - V(s))]$
        s←s'
    Until s is terminal

Figure 1.3. TD(0) Algorithm

TD(0) algorithm is run until a successful policy is found.

1.1.3. Q-Learning

One of the most important breakthroughs in reinforcement learning was the development of *Q-learning* (Watkins 1989). In Q learning, updating process is done

for action values instead of the state values. Best action of the next state is used as reward expectation in update process. In this way, a valuable action owned by a bad valued state is not ignored on the evolution progress. The update process of one-step Q-learning is done by;

$$Q(s_t, a_t) = Q(s_t, a_t) + \propto \left[r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t) \right] \qquad (1.6)$$

$Q(s_t, a_t)$ 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}. $\propto$ is a constant that specify how quickly the action values will change. γ is another constant that specifies how much would be the effect of the next best action's value on the previous action's value.

Initialize Q(s, a) arbitrarily
Repeat (for each episode):
    Initialize s
    Repeat (for each step of episode)
        Choose a from s using policy derived from Q
        Take action a, observe r, next state s'
        $Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma \max_{a'} Q(s', a') - Q(s, a))]$
        $s \leftarrow s'$
    Until s is terminal

Figure 1.4. Q-learning algorithm

In Q-learning algorithm for each episode, selected action values are updated with equation 1.6. After a successful policy is found, evolution process is finished.

The weakness of algorithm is need the storage of Q-values 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

millions of episodes to find a successful policy. In this case a generalisation method is needed to suit a successful policy for the problem.

1.1.4. Eligibility Traces

Eligibility traces is one of the basic methods that is used to make reinforcement learning methods more efficient. An eligibility trace keeps a record of an event, such as visiting a state or taking an action. After a reward is gained or a cost is spent, only eligible states or actions are awarded due to the reward or blamed due to the cost that they bring out.

1.1.4.1. N-Step Td Prediction

Tabular TD methods use next state values as the reward expectation in update process. n-step TD method considers next n actions' rewards and value of the last state which is reached as a result of the n actions, in order to update the state values. While one-step update value is

$$R_t^{(1)} = r_{t+1} + \gamma V_t(s_{t+1}), \tag{1.7}$$

two-step update value is

$$R_t^{(2)} = r_{t+1} + \gamma r_{t+2} + \gamma^2 V_t(s_{t+2}). \tag{1.8}$$

As a result, n-step update value will be

$$R_t^{(n)} = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots + \gamma^{n-1} r_{t+n} + \gamma^n V_t(s_{t+n}). \tag{1.9}$$

$R_t^{(n)}$ value is referred as *n-step return value at time t*. After each action, updating of state values is done with $\Delta V_t(s_t)$.

$$\Delta V_t(s_t) = \alpha\left[R_t^{(n)} - V_t(s_t)\right] \tag{1.10}$$

γ is a discount rate parameter which is relevant to rewards. There is also need a parameter to decide how much a reward or a cost must be affected to previous events, states or actions. This parameter is called as trace-decay parameter and is shown with λ. Eligibility trace values are updated after each action or state change.

$$e_t(s) = \gamma\lambda e_t(s) \qquad \text{if } s \neq s_t \tag{1.11}$$

$$e_t(s) = \gamma\lambda e_t(s) + 1 \qquad \text{if } s = s_t \tag{1.12}$$

Each state has its own trace value depends to the distance between the owner state and current state. Therefore, when a prize is gained every state is updated depend to its trace value.

Initialize V(s) arbitrarily, e(s)=0 for all sϵS (All eligibility traces are set to zero.)
Repeat (for each episode):
    Initialize s
    Repeat (for each step of episode)
        a← action given by π for s
        Take action a; observe reward, r, and next state, s'
        $\delta \leftarrow r + \gamma V(s') - V(s)$,
        $e(s) \leftarrow e(s) + 1$
        For all s:
        $V(s) \leftarrow V(s) + \alpha\delta e(s)$
        $e(s) \leftarrow \gamma\lambda e(s)$
    $s \leftarrow s'$
    Until s is terminal

Figure 1.5. TD(λ) Algorithm

1.1.4.2. N-Step Q-Learning

Difference of n-step Q-Learning is use of the action values rather than state values and use of the best next state action as reward expectation to update the action values. In this method $R^{(n)}$ is calculated as:

$$\boldsymbol{R_t^{(1)}} = r_{t+1} + \gamma \max_a Q_t(s_{t+1}, a)$$

$$\boldsymbol{R_t^{(2)}} = r_{t+1} + \gamma r_{t+2} + \gamma^2 \max_a Q_t(s_{t+2}, a)$$

...

$$\boldsymbol{R_t^{(n)}} = r_{t+1} + \gamma r_{t+2} + \cdots + \gamma^n \max_a Q_t(s_{t+n}, a) \quad (1.13)$$

Update of the action values is calculated as:

$$\Delta Q_t(s_t, a_t) = \alpha[R_t^{(n)} - Q_t(s_t, a_t)] \quad (1.14)$$

$$Q_{t+1}(s, a) = Q_t(s, a) + \Delta Q_t(s, a) \quad (1.15)$$

Initialize Q(s, a) arbitrarily, e(s, a)=0 for all s, a

Repeat (for each episode):

Initialize s, a

Repeat (for each step of episode)

Take action a, observe r, s'

Choose a' from s' using policy derived from Q (such as ε-greedy)

$a^* = arg\max_b Q(s', b)$

$\delta \leftarrow r + \gamma Q(s', a^*) - Q(s, a)$,

$e(s, a) \leftarrow e(s, a) + 1$

For all s, a:

$Q(s,a) \leftarrow Q(s,a) + \alpha\delta e(s,a)$

If$(a^* = a')$ then $e(s) \leftarrow \gamma\lambda e(s)$

Else $e(s) \leftarrow 0$;

$s \leftarrow s', a = a'$

Until s is terminal

Figure 1.6. Watkins's Q(λ) Algorithm

Watkins's Q(λ) algorithm keeps eligibility trace records for each visited state till best policy is followed. If an exploration (random) action is chosen depend to evaluation method, all eligibility traces are zeroed. Disadvantage of this algorithm is cutting off traces after each exploring action. Peng's Q(λ) algorithm (1996) is another form of this algorithm which can be used to prevent this disadvantage.

Initialize Q(s,a)=0, e(s,a)=0 for all s,a

Repeat (for each episode):

Initialize s,a

Repeat (for each step of episode)

Take action a, observe r, s'

Choose a' from s' using policy derived from Q (such as ε-greedy)

$a^* = arg\max_b Q(s', b)$

$\delta' \leftarrow r + \gamma V(s') - Q(s, a)$,

$\delta \leftarrow r + \gamma V(s') - V(s)$,

For all s, a:

$e(s) \leftarrow \gamma\lambda e(s)$

$Q(s,a) \leftarrow Q(s,a) + \alpha\delta e(s,a)$

$Q(s,a) \leftarrow Q(s,a) + \alpha\delta'$

$e(s,a) \leftarrow e(s,a) + 1$

$s \leftarrow s', a = a'$

Until s is terminal

Figure 1.7. Peng's Q(λ) Algorithm

In Peng's Q(λ) algorithm (Peng and Williams, 1996) eligibility traces are kept and updated till episode is finished. It is shown that this algorithm is also effective on reinforcement learning problems.

1.2. General Regression Neural Network

GRNN (Specht, 1991) is a memory-based neural network which does not require an iterative training that simply keeps all training data as pattern and forecasts value of an input data with help of them.

GRNN is a four layered network. First layer is input layer that contains a neuron for each input value. Second layer is pattern layer that keep 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 pass through the Gaussian activation function. Third layer which is called summation layer has two neurons that are named numerator and denominator. In this way numerator calculates a dot product between outputs of pattern layer and a vector which contains expected values of each training data. Denominator calculates sum of the outputs of pattern layer. At the end output layer calculates quotient of numerator and denominator values and finds the estimated value.

$$Y(x) = \frac{\sum_{i=1}^{n} Y_i \exp(-D_i^2/2\sigma^2)}{\sum_{i=1}^{n} \exp(-D_i^2/2\sigma^2)} \tag{1.16}$$

$$D_i^2 = (x - x_i)^T (x - x_i) \tag{1.17}$$

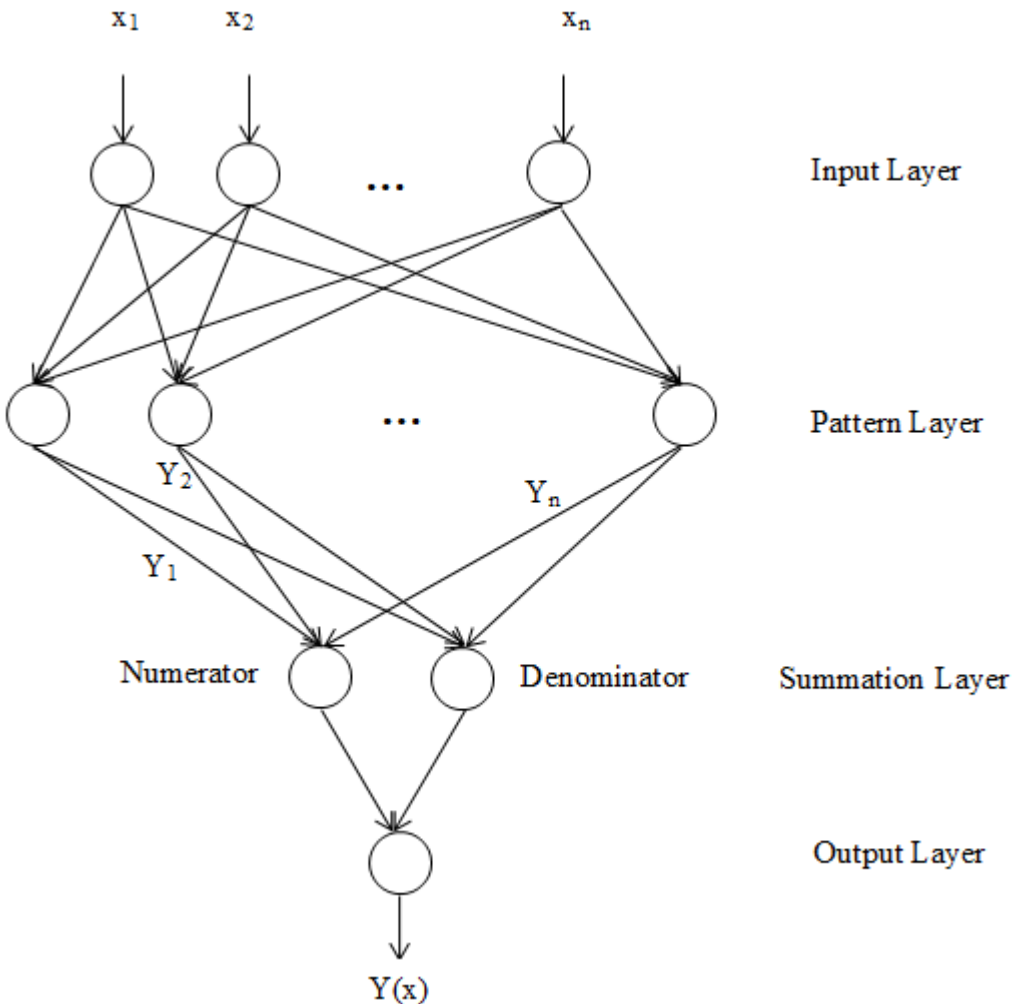


Figure 1.8. General Regression Neural Network Structure

1.3. Rl-Glue

All benchmarks are coded with C programming language on RL-Glue (Reinforcement Learning Glue) interface which is a standard interface that allows programming environment, agent and experiment separately and provides them to connect each other and works correctly. (Tanner and White, 2009)

2. PREVIOUS WORKS

Q learning is a breakthrough in reinforcement learning. Because of the large dimensional problems have huge amount of state-action pairs, Q learning are not so effective in many real-world problems. This problem is called “curse of dimensionality” by Sutton (1996). Millions of action selection can be needed to find a successful policy. In this case a regression method is needed. However, Q-value estimates never become exact values. This led many regression methods to be tried in reinforcement learning.

Anderson (1987) used one-layer and two-layer networks with error back-propagation to generalize value function and showed his own two-layer network design is able to solve pole-balancing task for TD learning. Lin also used back-propagation for generalization and applied it on different frameworks for TD learning and Q-Learning (1991, 1992).

Boyan and Moore (1995) concerned about hardness of robust generalization of value function with a function approximator such as a neural network. They proposed Grow-Support algorithm to prevent bad convergence in Dynamic Programming. Grow-Support algorithm suggests to select some state values from state space, in order to finds their expected cost amount with an iterative procedure. This procedure starts from chosen state and go through the episode until episode is finished or a defined cost limit is exceeded. This progress is done for all chosen states. After this, state values are updated. Then, a function approximator is used to generalize value function. After that, all progress is repeated with the new value function. It is repeated until the value function stops changing.

Sabes (1994) suggested using a basis function to represent value function in Q-Learning.

Gordon (1995) proposed using k-nearest-neighbour method to generalise value function. Determined number of states is kept to use for calculations of k-nearest-neighbour algorithm.

Sutton (1996) suggested sparse-coarse-coded function approximators (CMACs) to avoid poor performance.

Bradtke and Barto (1996) defined Least Square TD and Recursive Least Square TD algorithms. In the algorithms a linear approximation function is used with observed inputs and observed outputs. Outputs are defined by;

$$\psi_t = \Psi(\omega_t) + \eta_t \tag{2.1}$$

$$\psi_t = \omega_t'\theta + \eta_t \tag{2.2}$$

where Ψ is the linear function to be approximated, $\omega_t$is observed input at time t, ψ_t is the observed output at time t, η_t is output noise at time t and θ is the parameter vector. Sum of the least-squares between observed outputs and expected outputs can be written as;

$$J_t = \frac{1}{t}\sum_{k=1}^{t}[\psi_k - \omega_k'\theta_t]^2 \tag{2.3}$$

Partial derivative is taken with respect to θ_t. In order to equation is set to zero and solved to find θ parameter vector that minimizes error between observed and expected outputs.

$$\theta_t = \left[\frac{1}{t}\sum_{k=1}^{t}\omega_k\omega_k'\right]^{-1}\left[\frac{1}{t}\sum_{k=1}^{t}\omega_k\psi_k\right] \tag{2.4}$$

State value V can be defined as linear by;

$$V(x) = \Phi_x'\theta \tag{2.5}$$

where Φ_x is state vector and θ is parameter vector. As mentioned before, bellman equation which used as value function;

$$V(x) = \sum_{y \epsilon X} P(x,y)[R(x,y) + \gamma V(y)] \tag{2.6}$$

can be written as;

$$V(x) = \sum_{y \epsilon X} P(x,y)R(x,y) + \gamma \sum_{y \epsilon X} P(x,y)V(y) \tag{2.7}$$

$$V(x) = \hat{\mathrm{r}}_x + \gamma \sum_{y \epsilon X} P(x,y)V(y) \tag{2.8}$$

where $\hat{\mathrm{r}}_x$ is the expected immediate reward from any state transition from x. If V(x) and V(y) are replaced with 5th formula;

$$\hat{\mathrm{r}}_x = V(x) - \gamma \sum_{y \epsilon X} P(x,y)V(y) \tag{2.9}$$

$$\hat{\mathrm{r}}_x = \Phi_x'\theta - \gamma \sum_{y \epsilon X} P(x,y)\Phi_y'\theta \tag{2.10}$$

$$\hat{\mathrm{r}}_x = \left(\Phi_x' - \gamma \sum_{y \epsilon X} P(x,y)\Phi_y' \right)\theta \tag{2.11}$$

Reward function depending on time becomes;

$$r_t = \left(\Phi_t - \gamma \sum_{y \epsilon X} P(x,y)\Phi_y \right)' \theta \tag{2.12}$$

To maximize reward, derivative of least-squares could be used. If (2.4) is rewritten;

$$\theta_t = [\frac{1}{t}\sum_{k=1}^{t} \Phi_k(\Phi_k - \gamma\Phi_{k+1})']^{-1}\left[\frac{1}{t}\sum_{k=1}^{t} \Phi_k r_k\right] \quad (2.13)$$

This formula can be used for episode-based problems as;

```
Set t=0,
repeat forever {
        Set x_t to be a start state with respect to S probabilities.
        While x_t is not an end state {
                x_t = x_{t+1}
                Use (13) to define θ_t
                t = t + 1;
        }
}
```

Figure 2.1. Episode-Based LS TD

Bradtke and Barto (1996) also defined recursive version of LS TD algorithm and provided convergence proofs of the algorithms. Boyan (2002) updated LSTD algorithm in a way which gives ability to work with eligibility traces and named it LSTD(λ). Lagoudakis and Parr (2003) proposed Least Squares Policy Iteration method which able to learn state-action value function, means it can be used for Q learning.

Tsitsiklis and Van Roy (1997) analysed convergence of linear and non-linear function approximators for TD learning.

Riedmiller (1999) analysed concepts of neural control architecture which has ability to learn control behaviour in technical process control accurately.

Ernst, Geurts and Wehenkel (2005) suggested Fitted Q Iteration algorithm. This algorithm works offline means after each episode is finished entirely, obtained

experience values are used in regression algorithm. Experience values are kept in four-tuples form as *(s,a,r,s′)*. Here, s is the transition state, *a* is the selected action, *r* is reward and s' is the resulting state of action *a*.

Inputs: a set of four tuples F and a regression algorithm

Initialization:

Set N to 0

Let Q_N be a function equal to zero for all state, action pairs.

Iterations:

Repeat until stopping conditions are reached

- N++
- Build the trainin set $TS = \{(i^l, o^l), l = 1, \dots, \#F\}$ based on the function Q_{N-1} and on the full set of four-tuples F:

$$i^l = (s_t^l, a_t^l),$$

$$o^l = r_t^l + \gamma \max_{u \epsilon U} Q_{N-1}(s_{t+1}^l, a).$$

- Use the regression algorithm to induce from TS the function $Q_N(s, a)$

Figure 2.2. Fitted Q Iteration algorithm

As a result of the algorithm Q_N becomes;

$$Q_N(s,a) = r(s,a) + \gamma \max_{u' \epsilon U} Q_{N-1}(s', a') \tag{2.14}$$

After the distance between Q_N and Q_{N-1} becomes meaningless degree less, Q_N could be used is a regression method. Ernst, Geurts and Wehenkel also proposed tree based methods and informed about their performance.

Riedmiller (2005), taking the inheritance of Fitted Q algorithm, proposed NFQ algorithm which represent Q-value function with a multilayer perceptron network. NFQ algorithm works offline means after each episode is finished entirely, obtained experience values are used to update the network. Experience values are kept in triple form as (*s*, *a*, s'). Here, s is the transition state, *a* is the selected action and s' is the resulting state of action *a*.

Inputs: a set of transition samples D; output: Q-value function Q_N

k=0

Do {

Generate pattern set $P = \{(input', target'), l = 1, \dots, \#D\}$ where

$input' = s', a',$

$target' = c(s', a', s^n) + \gamma \min_{a^n} Q_k(s^n, a^n)$

$Q_{k+1} \leftarrow$ Rprop_training(P)

k++

}while(k<N)

Figure 2.3. NFQ algorithm

Rprop which is used in algorithm is a fast back-propagation algorithm proposed by Riedmiller and Braun (1993). c is the cost function. Target value is determined with;

$$target^l = \begin{cases} c(s', a', s^n), if\ s^n \epsilon S^+ \\ C^-, if\ s^n \epsilon S^- \\ c(s', a', s^n) + \gamma \max_{a^n} Q_k(s^n, a^n), else \end{cases} \quad (2.15)$$

S^+ shows goal states. S^- shows forbidden states which must be avoided by a successful policy. C^- is equal to 1, maximum value for multilayer perceptron.

Whiteson (2006) proposed to use NeuroEvoulotion of Augmenting Topologies (NEAT) which suggested by Stanley and Miikkulainen (2002) to approximate Q value function. NEAT-Q also is a method that uses back propagation to convergence the value function.

Cetina (2008) proposed to use multilayer perceptron with radial basis functions as Q-value function. It is mentioned about using a radial basis function layer is prevented to stuck in local minimum.

Shibuya (2010) suggested to use Complex-values RBF network to generalize Q-value function.

3. PROPOSED METHOD AND TESTING BENCHMARKS

3.1. Proposed Method

3.1.1. QRNN Methodology

Q Learning was important newness for reinforcement learning. However huge size of state-action space of the real world problems makes Q learning inapplicable and unworkable. An efficient regression method is needed for not only to generalize the state, action space but also accelerate speed of learning algorithm.

Suggested reinforcement general regression neural network is a GRNN form neural network which is used to generalize Q-value function. Main difference between QRNN and other neural networks which have been used to generalize Q-value function is QRNN is not an error-backpropagation network. Therefore QRNN does not need a training phase. QRNN needs mostly accurate data in the pattern layer to be able to predict the expected reward of a state-action pair. In this case, Q-Learning algorithm is used to generate a bunch of episodes. While Q-Learning algorithm is working, QRNN is established with the selected state-action pairs and target values of them.

$$\text{target} = \text{r} + \gamma \max_{a'} Q(s', a') \tag{3.1}$$

To limit pattern layer size of the network, state-action space is divided into finite number of pieces and then each piece of the space is represent by a pattern neuron. When a new triple (s, a, target) is observed, firstly it is checked to know which pattern neuron is responsible for it. If this neuron does not exist, the neuron is added to the pattern layer. If exists, neuron values are updated with new triple.

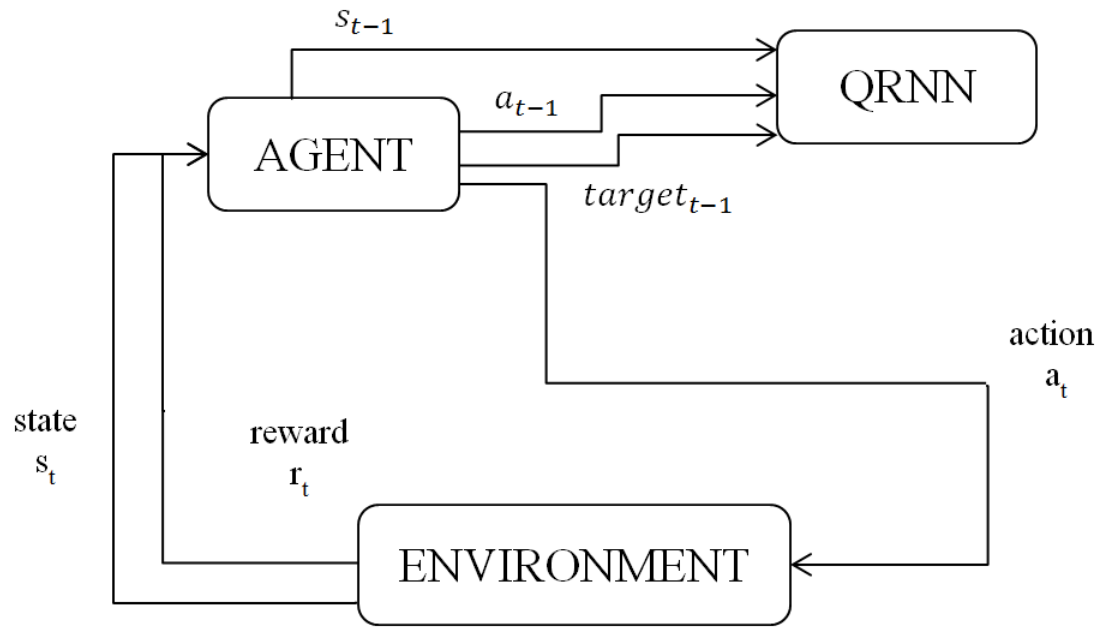


Figure 3.1. Training of QRNN

After a bunch of episodes are experienced, QRNN's effectivity is tried on the same problem. For each action, current state parameters are passed through the QRNN and action has highest reward expectation is selected.

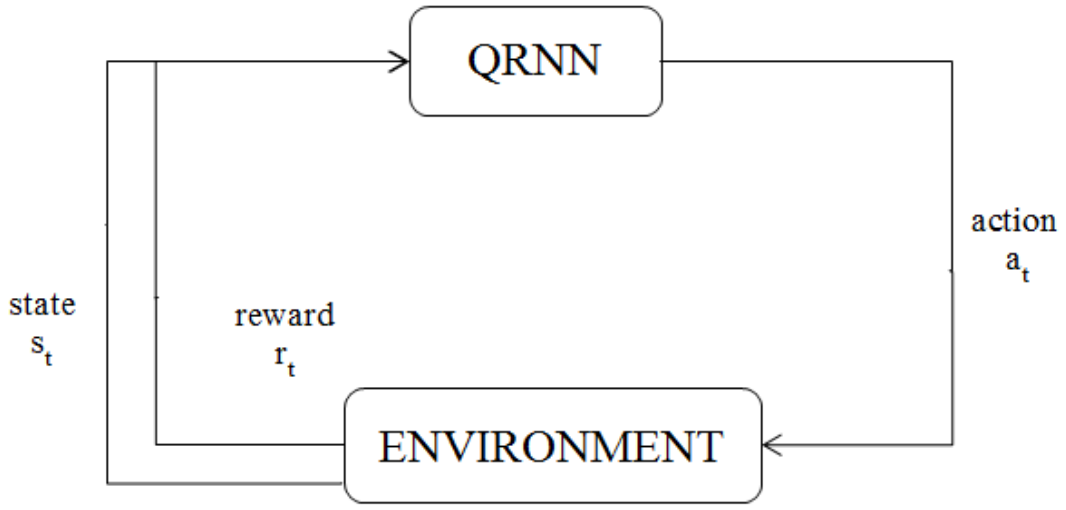


Figure 3.2. Usage of QRNN

3.1.2. QRNN Structure

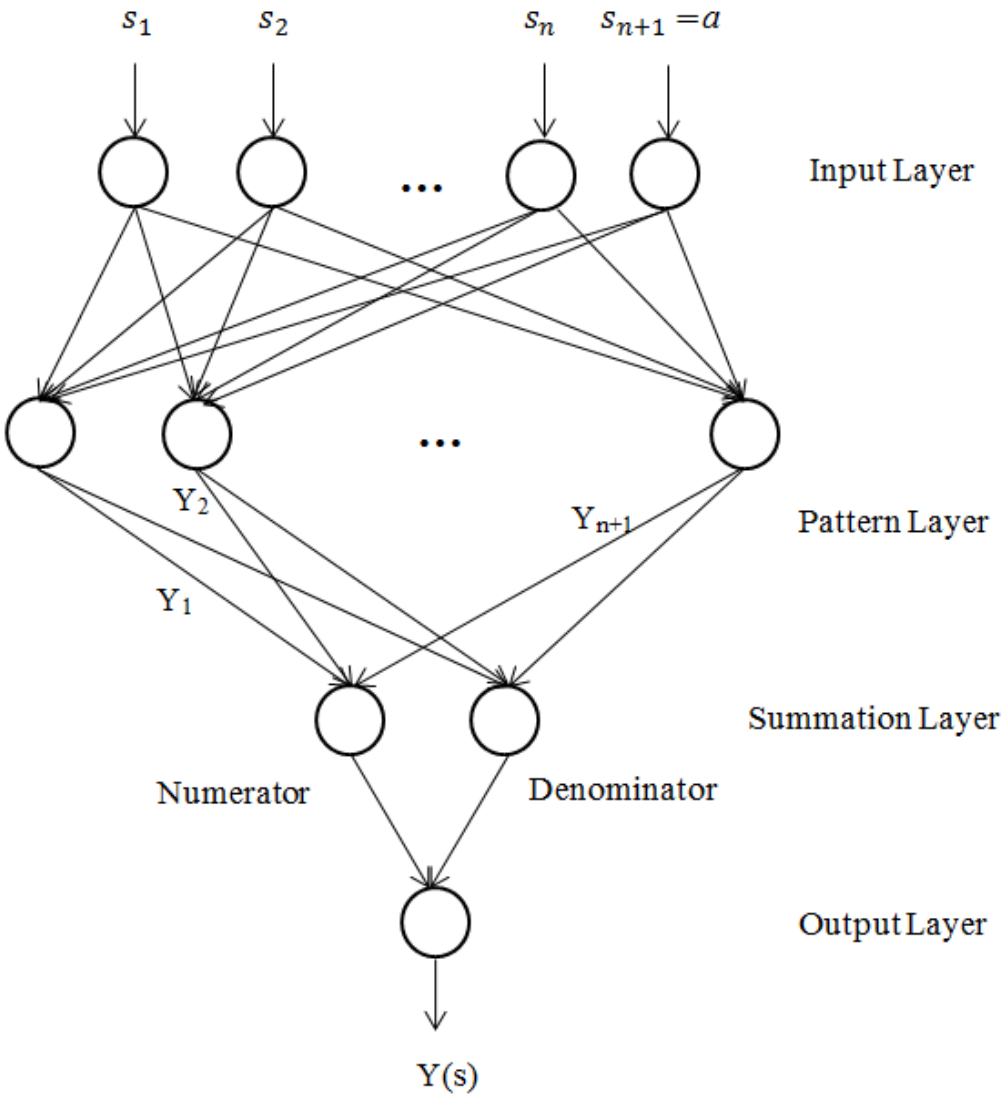


Figure 3.3. QRNN Structure

$$D_i^2 = (s - s_i)^T (s - s_i) w^T w \tag{3.2}$$

$$Y(s) = \frac{\sum_{i=1}^{n+1} Y_i \exp(-D_i^2 / 2\sigma^2)}{\sum_{i=1}^{n+1} \exp(-D_i^2 / 2\sigma^2)} \tag{3.3}$$

s is a vector that keeps all features of a state. a is the action value. *w* is a vector keeps sensitivity variables which are used to decide which parameter is

prescriptive. More clearly, if *w* parameter of a state is higher than other parameters of the state, this causes this parameter to become less effective on the evolution progress. σ is variance of the distribution. *w* and σ have opposite effect on a parameter.

3.1.3. QRNN Algorithm

Initialize $Q(s, a)$=0
Repeat forever:
  Repeat (for n episode):
    Initialize s
    Repeat (for each step of episode)
    Choose a from s using policy derived from Q
    Take action a, observe r, next state s'
    $Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma \max_{a'} Q(s',a') - Q(s,a))]$
    $target = r + \gamma \max_{a'} Q(s',a')$
    If (s, a) is exist in pattern layer of QRNN
      Update pattern neuron with (s, a) and target values
    else
      Add a new neuron to QRNN with (s, a) and target values
    $s \leftarrow s'$
    Until s is terminal
Measure QRNN efficiency with k random episodes
If QRNN is efficient break

Figure 3.4. QRNN algorithm

In QRNN algorithm, neural network creation and Q evolution process are done simultaneously. While Q learning algorithm is run, data created by Q agent is used to establish QRNN. For each action taken by the agent, a neuron is added or updated (if (s, a) is exists in pattern layer) with state, action and also calculated target

values. After each n episodes, efficiency of QRNN is measured by k random episodes. If QRNN is efficient, evolution progress is done.

3.2. Training Benchmarks

Four different benchmarks are used to measure quality of QRNN generalization. Random walk is the simplest one which has a simple Q-value distribution. Other benchmarks are mountain-car, 2-parameter pole-balance and 4-parameter pole-balance environments which are commonly used to measure performance of reinforcement learning algorithms.

3.2.1. Random Walk

One dimensional random walk is one of the benchmarks that is used to measure quality of a reinforcement learning algorithm. There are five states on a line and each state connects to other two states which one of them is in the left, and other one is on the right. Each state has two possible actions to be taken, “left” or “right”. Each action has same possibility (%50). There is only one reward to be won; this reward is gained when the episode is finished with the taking “right” action on the fifth state to terminal. There is also another terminal state at the left of the first state. It is recognized easily, best policy to be followed is consistently going to right for each state.

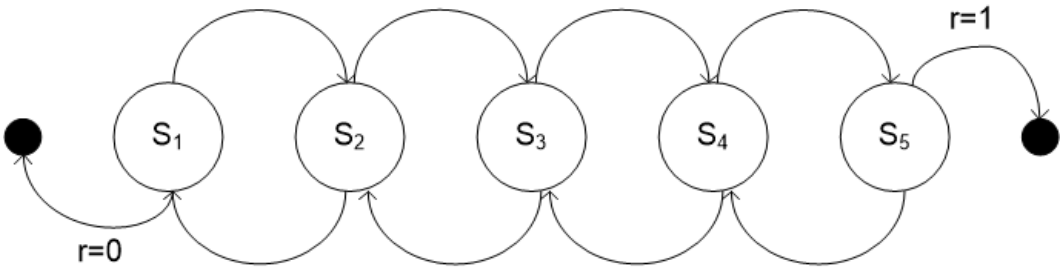


Figure 3.5. Random Walk Benchmark

3.2.2. Mountain Car

The mountain car problem is firstly defined by Moore in his PhD Thesis (1990). When Singh and Sutton added the problem into "Reinforcement Learning: An Introduction" (1998) book, it became more popular. The problem is about a powerless car must overcome a hill. Due to the gravity is stronger than the car's engine; the car can't 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.

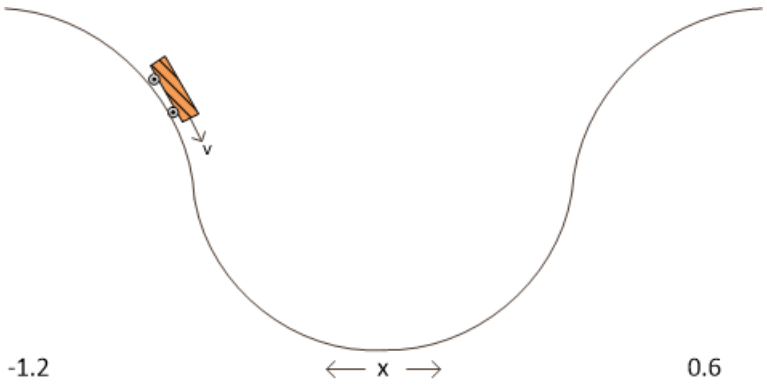


Figure 3.6. Mountain-Car Benchmark

Environmental parameters:

$$Velocity = (-0.07, +0.07)$$
$$Position = (-1.2, 0.6)$$
$$Actions = (-1, +1)$$

For each time step reward is -1. After each action parameter changes according to this formulas;

$$Velocity \mathrel{+}= Action * 0.001 + \cos(3 * Position) * (-0.0025) \quad (3.4)$$

$$Position = Position + Velocity \quad (3.5)$$

As a starting condition, position is chosen randomly and velocity is set to zero. When position becomes higher then 0.6, episode is terminated.

3.2.3. Pole-Balance Benchmarks

Pole-balance is a classic problem is widely used for testing control algorithms.

3.2.3.1. 2-Parameter Benchmark

In two-dimensional world, a cart with a pole onto is places on a track. There is three actions (-50N, 0N, +50N). Two of the actions force the cart with 50N to opposite directions, Third one doesn't affect the cart. Goal is to prevent the pole to fall.

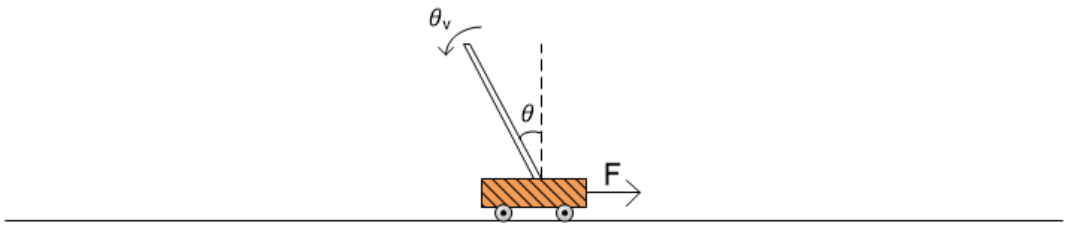


Figure 3.7. Pole-Balance(Inverted-Pendulum) 2-Parameter Benchmark

There is only one negative reward (-1) which is gained when the pole falls. There are two parameter associated with the problem: θ is position of the pole as radian value, θ_v is angular velocity of the pole. Value range of the parameters:

$$\theta \epsilon \left[-\frac{\pi}{2}, \frac{\pi}{2}\right] rad$$

$$\theta_v \epsilon [-2.5, 2.5] rad/s$$

Each action changes acceleration of the cart and the pole according to:

$$\theta_a(t) = \frac{gsin\theta(t) - cos\theta(t)\left(\frac{F(t) + ml\theta_v(t)^2 sin\theta(t)}{m_c + m}\right)}{l\left(\frac{4}{3} - \frac{mcos^2\theta(t)}{m_c + m}\right)} \tag{3.6}$$

where $g = 9.8\, m/s^2$ is the gravity, $l = 0.5m$ is the half-pole length, $\mathrm{F(t)} = \mathrm{F} + \eta$ is the force where η is the noise term valued in range of [-10,10], $m = 2Kg$ and $m_c = 8Kg$ are the masses of the pole and the cart.

In order to positions and velocities of the cart and the pole is changed by the following equations:

$$\theta(t) = \theta(t) + \tau * \theta_v(t) \tag{3.7}$$

$$\theta_v(t) = \theta_v(t) + \tau * \theta_a(t) \tag{3.8}$$

$\tau = 0.1\mathrm{s}$ is used as time step for each movement.

3.2.3.2. 4-Parameter Benchmark

In this case, QRNN is tried in a harder environment. QRNN must not only learn to balance the pole and also must prevent the cart quit from the space reserved, and also angular range which the pole must be in is much more smaller in this environment. There is two actions (-10N, 10N). Two of the actions force the cart with 10N to opposite directions, Goal is to prevent the pole to fall down and also prevent the cart to get out of the field.

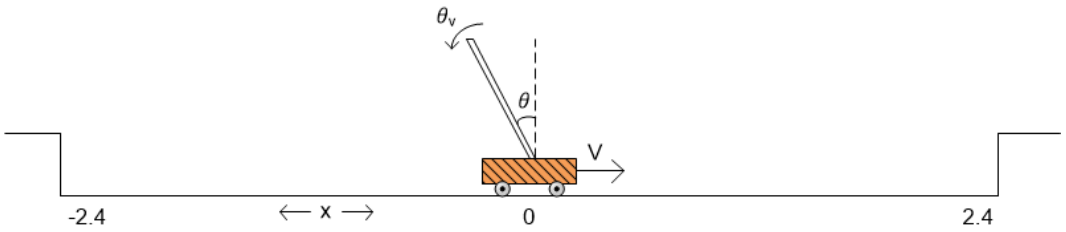


Figure 3.8. Pole-Balance(Inverted-Pendulum) 4-Parameter Benchmark

There is only one negative reward (-1) which is gained when the pole falls or cart gets out of the area. There are four parameter associated with the problem: x is the position of the cart, v is velocity of the cart, θ is position of the pole as radian value, θ_v is angular velocity of the pole. Value range of the parameters:

$x\epsilon[-2.4,2.4]m$

$v\epsilon[-1,1]m/s$

$\theta\epsilon\left[-\frac{\pi}{2},\frac{\pi}{2}\right]rad$

$\theta_v\epsilon[-2.5,2.5]rad/s$

Each action changes acceleration of the cart and the pole according to:

$$\theta_a(t)=\frac{gsin\theta(t)-cos\theta(t)\left(\frac{F(t)+ml\theta_v(t)^2sin\theta(t)}{m_c+m}\right)}{l\left(\frac{4}{3}-\frac{mcos^2\theta(t)}{m_c+m}\right)} \tag{3.9}$$

$$a(t)=\frac{\left(\frac{F(t)+ml\theta_v(t)^2sin\theta(t)}{m_c+m}\right)-ml\theta_a(t)cos\theta(t)}{m_c+m} \tag{3.10}$$

where $g=9.8\,m/s^2$ is the gravity, $l=0.5m$ is the half-pole length, F(t) = F + η is the force where η is the noise term valued in range of [-1,1], $m=0.1Kg$ and $m_c=1Kg$ are the masses of the pole and the cart.

Positions and velocities of the cart and the pole is changed by the following equations:

$$x(t+1) = x(t) + \tau * v(t) \tag{3.11}$$

$$v(t+1) = v(t) + \tau * a(t) \tag{3.12}$$

$$\theta(t) = \theta(t) + \tau * \theta_v(t) \tag{3.13}$$

$$\theta_v(t) = \theta_v(t) + \tau * \theta_a(t) \tag{3.14}$$

$\tau = 0.02\text{s}$ is used as time step for each movement.

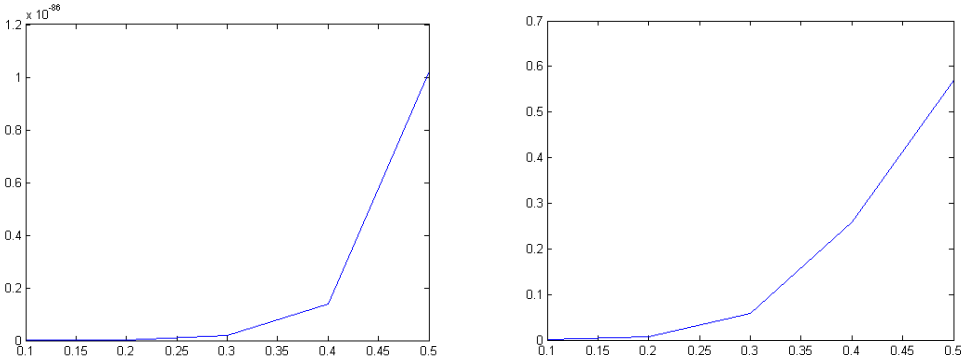
4. PERFORMANCE ANALYSIS AND FUTURE WORK

4.1. Performance Analysis Results

4.1.1. Random Walk Results

Just one successful episode is enough for QRNN to learn best policy for random walk. After following episode, values of the actions are shown.

$$[(s_3, left), (s_2, right), (s_3, right), (s_4, right), (s_5, right)]$$



Left action Right Action

Figure 4.1: Left and right action values calculated by QRNN for random walk

In the figure x axis shows states. There are five states that are valued between 0.1 and 0.5 with 0.1 sensitivity. y axis shows expected reward of the action. As mentioned before, best policy is "*right* action" for each state. It is seen expected reward of *right* action is higher for all states.

4.1.2. Mountain-Car Benchmark Results

State-action space is defined in the size of $1800 * 14 * 2$. 1800 different position values, 14 different velocity values and 2 different actions are used. Therefore maximum number of pattern nodes is 50400. Most of the nodes is not created because of unvisited or irrational state-action pairs during the episodes. So

number of nodes in the pattern layer become furthest about 18000. Training episodes starts with a random position and zero velocity. Maximum step size of a training episode is limited with 5000. After each 25 episode QRNN performance is measured with different variance values between 0.05 and 0.5 for different 180 states on the benchmark. Testing state positions starts from -1.2 and are increased by 0.01 till 0.59. *w* is set to 1 for all state parameters.

QRNN is trained and performance of QRNN is measured with given parameters for 10 times. Test episodes are limited with maximum number of 5000 steps. When QRNN become entirely successful on all test episodes, best Q-agent policy efficiency is measured and test results is shown at the Table 4.1.

Table 4.1. Mountain-car experiment results

Test No	**Training Episodes Length**	**Best Q-Agent Policy Efficiency**	**QRNN Performance**	**QRNN Variance**	**QRNN Min-Max Episode Step Size**
1	25	11/180 (%6.1)	180/180(%100)	0.05	(8, 216)
2	25	16/180 (%8.8)	180/180(%100)	0.175	(40, 212)
3	50	18/180 (%10.0)	180/180(%100)	0.40	(3, 222)
4	275	36/180 (%0.0)	180/180(%100)	0.275	(3, 185)
5	125	21/180 (%11.6)	180/180(%100)	0.20	(3, 213)
6	225	34/180 (%18.8)	180/180(%100)	0.30	(3, 209)
7	75	8/180 (%4.4)	180/180(%100)	0.125	(40, 201)
8	150	23/180 (%12.7)	180/180(%100)	0.175	(41, 181)
9	25	16/180 (%8.8)	180/180(%100)	0.100	(3, 229)
10	25	18/180 (%10.0)	180/180(%100)	0.225	(40, 211)
Ave	100	20/180 (%11.1)	180/180(%100)	-	-

It is obtained that, while Q-learning needs more than 10000 episodes to learn a successful policy, QRNN requires only averagely 100 episodes. Also, QRNN has much better and robust results than Q-learning.

4.1.3. Pole-Balance Benchmark Results

4.1.3.1. 2-Parameter benchmark

In two-parameter benchmark only θ and θ_v are used as state parameters. State-action space is in size of $32 * 50 * 3$. Maximum number of pattern nodes becomes 3200. There are unreasonable states which can't be visited, in case number of nodes in the pattern layer become maximum about 1500. Training episodes starts with $\theta = 0$ and $\theta_v = 0$. Maximum step size of a training episode is limited with 3000, each movement takes 0.1 sec, which is equal to 300 second (5 minute) of movement. After each 10 episode QRNN performance is measured with different variance values between 0.02 and 0.2 for different 50 test episodes on the benchmark. *w* is set to 1 for all state parameters. It is obtained that averagely 500 episodes is enough to learn a successful policy for QRNN.

QRNN is trained and performance of QRNN is measured with given parameters for 10 times. Test episodes are limited with maximum number of 3000 steps. When QRNN become entirely successful on all test episodes, best Q-agent policy efficiency is measured and test results is shown at the Table 4.2.

Table 4.2. 2-Parameter Pole-Balance experiment results

Training No	Training Episodes	Best Q-Agent Policy Balancing Time	QRNN Balancing Time	QRNN Successful Variance
1	440 ep.	1.1 sec.	5min.	0.06
2	510 ep.	1.1 sec.	5min.	0.06 to 0.08 and 0.10 to 0.12
3	490 ep.	1.1 sec.	5min.	0.02 to 0.10
4	670 ep.	1.14 sec.	5min.	0.02 to 0.13
5	430 ep.	1.07 sec.	5min.	0.02 to 0.05
6	530 ep.	1.14 sec.	5min.	0.02 to 0.10
7	420 ep.	1.07 sec.	5min.	0.02 to 0.04
8	580 ep.	1.09 sec.	5min.	0.04 to 0.14
9	530 ep.	1.18 sec.	5min.	0.02, 0.05, 0.06, 0.08, and 0.09
10	530 ep.	1.05 sec.	5min.	0.02 to 0.12
Ave	513 ep.	1.1 sec	5min.	-

QRNN needs averagely 513 episodes to learn a successful policy. It was presented (Lagoudakis and Parr, 2003), LSPI can balance the pole about 285 second after 1000 episodes. Q learning with experience replay, suggested in same paper, requires averagely 700 episodes to learn a successful policy.

It was reported NFQ algorithm (Riedmiller, 2005) needs averagely 200 episodes to find a successful policy. But, each episode is repeated over the network 50 times to learn with backpropagation. In this case training of the network takes 10000 passes. In contrast, QRNN is not a type of backpropagation network. After averagely 513 episodes is run, establishment of the neural network is done and a successful neural network is ready to use.

4.1.3.2. 4-Parameter benchmark

In this case, any state is defined with four parameters $(\mathrm{x}, \mathrm{v}, \theta, \theta_v)$ and State-action space is in size of $10 * 10 * 11 * 10 * 2$. Number of nodes in the pattern layer become maximum about 5000. Training episodes starts with random x between -1 and 1,$v = 0$, $\theta = 0$ and $\theta_v = 0$. Maximum step size of a training episode is limited with 3000 which is equal to 60 second (1 minute) of movement. After each 50 episode QRNN performance is measured with different variance values between 0.02 and 0.5 for different 50 test episodes on the benchmark. w is set to 1 for all state parameters. It is obtained that averagely 2960 episodes is enough to learn a successful policy for QRNN.

QRNN is trained and performance of QRNN is measured with given parameters for 10 times. Test episodes are limited with maximum number of 3000 steps. Result are given in the Table 4.3.

Table 4.3. 4- Parameter Pole-Balance experiment results

No	Training Episodes	Best Q-Agent Policy Time Average	QRNN Time Average	QRNN Successful Variances
1	2500ep.	5.28sec.	1min.	0.136 to 0.174
2	600 ep.	0.48sec.	1min.	0.434 to 0.478
3	2900ep.	5.11sec.	1min.	0.111
4	3400ep.	8.45sec.	1min.	0.124 to 0.158
5	3700ep.	6.22sec.	1min.	0.130 and 0.131
6	3350ep.	10.0sec.	1min.	0.234
7	2850ep.	6.03sec.	1min.	0.142 to 0.180
8	3550ep.	6.42sec.	1min.	0.124 to 0.178
9	3600ep.	8.93sec.	1min.	0.104 to 0.108
10	3150ep.	5.66sec.	1min.	0.148 to 0.246
Ave	2960ep.	6.27sec.	1min.	-

It was reported NFQ algorithm (Riedmiller, 2005) needs averagely 14440 cycles to find a successful policy. QRNN needs averagely 13000 cycles (equals to 3000 episodes) to find a successful policy.

4.2. Future Work

QRNN is an effective method for many kind of problems. But as it can be considered, QRNN performance is directly related to Q-Values of the Q learning algorithm. This problem can be prevented by using QRNN not only a regression method but also a learning algorithm.

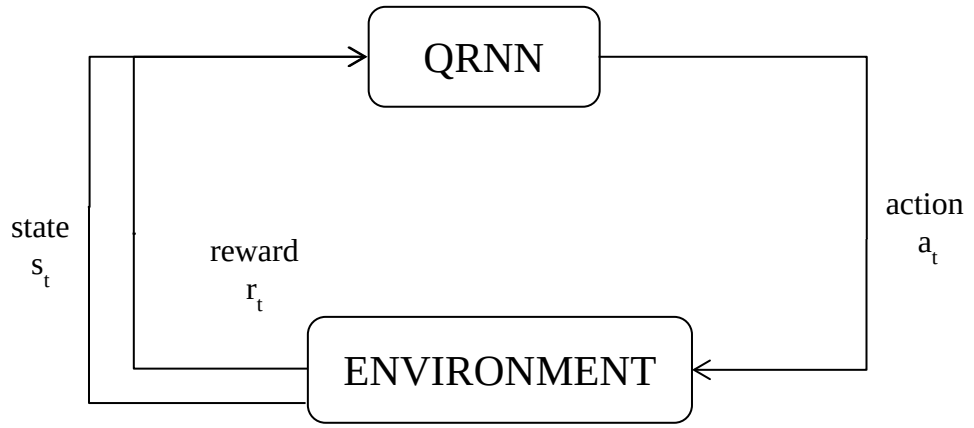


Figure 4.2: Training and usage of Improved QRNN

Repeat until stopping conditions are occurred:
n is a small integer;
    Repeat (for n episode):
    Empty Value_Array;
        Initialize s
        Repeat (for each step of episode)
        Choose best valued a from s using QRNN
        Take action a, observe r, next state s'
        Take $\max_{a'} Q(s', a')$ using QRNN
        target = r $+ \gamma \max_{a'} Q(s', a')$
        Keep s , a and *target* values in *Value_Array*
        $s \leftarrow s'$
        Until s is terminal
    For each s , a and *target* triples in *Value_Array*
        If s, a is exist in pattern layer of QRNN
            Update pattern neuron with s, a and target values
        else
            Add new neuron to QRNN with s, a and *target* values
Measure QRNN efficiency with k random episodes
If QRNN is efficient break

Figure 4.3: Improved QRNN Algorithm

As it is seen, QRNN learning algorithm is a batch learning method means values of state-action pairs are updated after the episode is done. Reason of this process is to prevent undesired local minimum and local maximum values on the regression surface.

According to the results obtained in first, learning algorithm learns faster than the previous methodology, but also unwanted oscillations can be occurred on the regression surface. Therefore the methodology is still in progress.

5. CONCLUSIONS

In this thesis, a very effective unsupervised learning regression neural network able to generalize the solution surface or solution hyper plane of reinforcement learning problems named as QRNN is proposed. The high efficiency of regression is obtained by the utilization of GRNN inherited network topology. On the other hand, main drawbacks of GRNN are also contained by QRNN. These are mainly problematic of the selection the efficient variance value for the corresponding data set and relatively high throughput time for a new data due to complex calculation. Estimation time consumption is decreased by limiting number of pattern layer nodes of the QRNN.

Under limited number of pattern layer neurons, QRNN is also effective with real time learning. It may be created and tested while Q agent is running on the environment. QRNN does not require an extra training time due to its state keeping structure. In backpropagation networks, a bunch of episodes must be stored as training data and then they must be passed through the network for many epochs. If learning process is unsuccessful, a new or bigger data set must be tried for the error back propagation networks. In contrast, QRNN evolution process can be done by growing the states kept in pattern layer. After a number of episodes used to establish QRNN, additional episodes can also be used to improve performance of the network. An iterative training process is not required for QRNN. Through this work QRNN is applied to solve popular problems of reinforcement learning that are random walk, mountain car and pole balance problems. Regression performance of QRNN is compared with that of LSPI and NFQ algorithms on the same test benches. QRNN learns much faster than these two algorithms. Tests are done by considering number of learning steps. In mountain car problem the learning is accelerated more than 100 times. In pole balance problem with two parameters; QRNN is faster approximately 1.4 times than LSPI and it is also faster approximately 19 times than NFQ. In pole balance problem with four parameters, QRNN is approximately 1.11 times faster than NFQ although QRNN is implemented in a wider state action space. These test results prove the efficiency and show importance of the proposed network.

REFERENCES

BARRETO, A. D. S., ANDERSON, C. W., 2007. Restricted gradient-descent algorithm for value-function approximation in reinforcement learning, Artificial Intelligence, March 2008, 172(4-5):454-482

BELLMAN, R. E. 1957. A Markov decision process, Journal of Mathematical Mech., 6:679-684

BELLMAN, R. E., 1957. Dynamic Programming, Princeton University Press, Princeton, NJ.

BERTSEKAS, D. P., 2010. Approximate Dynamic Programming, Dynamic Programming and Optimal Control II (3 ed.).

BOYAN, J. A., and MOORE, A. W., Generalization in reinforcement learning: Safely approximating the value function, Advances in Neural Information Processing Systems, vol. 7, 1995 :MIT Press.

BOYAN, J. A., Technical Update: Least-Squares Temporal Difference Learning, Machine Learning, v.49 n.2-3, p.233-246, November-December 2002

BRADTKE, S. J., BARTO, G. A., 1996. Linear least-squares algorithms for temporal difference learning, Machine Learning, v.22 n.1-3, p.33-57, Jan./Feb./March 1996 .

BRADTKE, S. J., ANDREW, G. B., 1996. Learning to predict by the method of temporal differences, Machine Learning (Springer) 22: 33–57.

CETINA, V. U., 2008. Multilayer Perceptron with Radial Basis Function as Value Function in Reinforcement Learning

ERNST, D., GEURTS, P., and WEHENKEL, L., 2005. Tree-based batch mode reinforcement learning. Journal of Machine Learning Research, 6, 503-556.

FAIRBANKS, M., ALONSO, E., 2012. The divergence of reinforcement learning algorithms with value-iteration and function approximation, In Proceedings of the 2012 International Joint Conference on Neural Networks (IJCNN), Brisbane, Queensland, Australia, 10–15 June (pp. 1–8). doi: 10.1109/IJCNN.2012.6252792.

GEIST, M., PIETQUIN, O., 2011. Parametric Value Function Approximation: a Unified View, ADPRL (2011).

GORDON, G. J., 1995. Stable function approximation in dynamic programming, Carnegie Mellon Univ.

KAELBLING, L. P. and MICHAEL L. L. and Andrew W. M., 1996. Reinforcement Learning: A Survey, Journal of Artificial Intelligence Research 4: 237–285.

KONIDARIS, G., 2008. Value function approximation in Reinforement Learning using the fourer basis, Computer Science Department Faculty Publication Series Paper 101.

LAGOUDAKIS, M. G., and PARR, R., 2003. Least-squares policy iteration, Journal of Machine Learning Research, 4, 1107–1149.

LANGLOIS, M. and SLOAN, R. H., Reinforcement learning via approximation of the Q-function, Journal of Experimental & Theoretical Artificial Intelligence Vol. 22, No. 3, September, 2010, 219-235.

LIN, L.J., 1992. Self-improving reactive agents based on reinforcement learning, planning and teaching Machine Learning, 8 (1992), pp. 293–321.

MCGOVERN, A. and SUTTON, R. S., 1997. Towards a better Q(lambda), Presented at the Fall 1997 Reinforcement Learning Workshop

MENACHE, I., MANNOR, S., and SHIMKIN N., 2005. Basis Function Adaption in Temporal Difference Reinforcement Learning, Annals of Operations Research 134, 215–238

PENG, J. and WILLIAMS, R. J., 1994. Incremental multi-step q-learning. In Cohen, W. W. and Hirsh, H., editors, Proceedings of the Eleventh International Conference on Machine Learning, pages 226-232.

POWELL, W., 2007. Approximate dynamic programming: solving the curses of dimensionality, Wiley-Interscience. ISBN 0-470-17155-3.

RIEDMILLER, M., 2000. Concepts and facilities of a neural reinforcement learning control architecture for technical process control, Journal of Neural Computing and Application 8, 323–338.

RIEDMILLER, M., 2005. Neural Fitted Q Iteration - First Experiences with a Data Efficient Neural Reinforcement Learning Method, J. Gama et al. (Eds.): ECML 2005, LNAI 3720, pp. 317–328, © Springer-Verlag Berlin Heidelberg 2005.

SABES, P., 1993. Approximating Q-values with basis function representations, Proceedings of the Fourth Connectionist Models Summer School, Lawrence Erlbaum, Hillsdale, NJ (1993).

SHIBUYA, T., and ARITA, H., HAMAGAMI, T., 2010. Reinforcement learning in continuous state space with perceptual aliasing by using complex-valued RBF network, IEEE International Conference on Systems, Man and Cybernetics, 2010, :1799-1803.

SINGH, S. P. and SUTTON, R. S., 1996. Reinforcement learning with replacing eligibility traces, Machine Learning, 22:123-158.

SPECHT, D.F., 1991. A General Regression Neural Network, IEEE transactions on neural networks, vol. 2. No. 6. page 568-576

SUTTON, R. S. and BARTO, A. G., 1998. Reinforcement Learning: An Introduction, MIT Press. ISBN 0-262-19398-1.

SUTTON, R. S. (1988). "Learning to predict by the method of temporal differences". *Machine Learning* (Springer) 3: 9–44.

SUTTON, R. S., 1996. Generalization in reinforcement learning: Successful examples using sparse coarse coding, In Advances in Neural Information Processing Systems 8: Proceedings of the 1995 Conference, pages 1038-1044, Denver, Colorado, 1996.

SUTTON, R. S., 1984. *Temporal Credit Assignment in Reinforcement Learning*. PhD thesis, University of Massachusetts, Amherst, MA.

SZITA, I., and LORINCZ, A., 2003. Reinforcement learning with linear function approximation and LQ control converges, Technical report, arXiv:cs/0306120v2.

TANNER, B., and WHITE, A., 2009. RL-Glue: Language-Independent Software for Reinforcement-Learning Experiments. Journal of Machine Learning Research, 10(Sep):2133--2136, 2009. (BibTex)

TESAURO, G., 1992. Practical issues in temporal difference learning. Mach. Learning 8, 257-277.

TSITSIKHS, J. N., and VAN ROY, B., 1997. An analysis of temporal-difference learning with function approximation, IEEE Transactions on Automatic Control 42: 674-690.

TSITSIKLIS, J., 1994. Asynchronous stochastic approximation and Q-learning, Machine Learning, 16(3) 185–202.

UENO, T., MAEDA, S.I., KAWANABE, M., Ishii, S., 2011. Generalized TD learning, Journal of Machine Learning Research, June 2011, 12:1977-2020

WATKINS, C. J. C. H., 1989. Learning from Delayed Rewards, PhD thesis, Cambridge University, Cambridge, England.

WATKINS, C. J. C. H., and DAYAN, P., 1992. Q-learning, Machine Learning, 8:279-292.

WHITESON, S., STONE, P., 2006. Evolutionary Function Approximation for Reinforcement Learning, The Journal of Machine Learning Research, 7, p.877-917, 12/1/2006.

BIOGRAPHY

Mehmet SARIGÜL was born in Hatay, in 1989. he completed his elementary education at General Refet Bele İlköğretim Okulu. He graduated from Harbiye Lisesi, in 2005. He received the B.E.degree from Department of Computer Engineering, Çukurova University in 2010. His research interests include machine learning, reinforcement learning, artificial neural network algorithms and applications, software design.