FPGA Accelerator Architecture for Q-learning and its Applications in Space Exploration Rovers

by

Pranay Reddy Gankidi

A Thesis Presented in Partial Fulfillment
of the Requirements for the Degree
Master of Science

Approved November 2016 by the
Graduate Supervisory Committee:

Jekanthan Thangavelautham, Chair
Fengbo Ren
Jae-sun Seo

ARIZONA STATE UNIVERSITY

December 2016

ABSTRACT

Achieving human level intelligence is a long-term goal for many Artificial Intelligence (AI) researchers. Recent developments in combining deep learning and reinforcement learning helped us to move a step forward in achieving this goal. Reinforcement learning using a delayed reward mechanism is an approach to machine intelligence which studies decision making with control and how a decision making agent can learn to act optimally in an environment-unaware conditions.

Q-learning is one of the model-free reinforcement directed learning strategies which uses temporal differences to estimate the performances of state-action pairs called Q values. A simple implementation of Q-learning algorithm can be done using a Q table memory to store and update the Q values. However, with an increase in state space data due to a complex environment, and with an increase in possible number of actions an agent can perform, Q table reaches its space limit and would be difficult to scale well. Q-learning with neural networks eliminates the use of Q table by approximating the Q function using neural networks.

Autonomous agents need to develop cognitive properties and become self-adaptive to be deployable in any environment. Reinforcement learning with Q-learning have been very efficient in solving such problems. However, embedded systems like space rovers and autonomous robots rarely implement such techniques due to the constraints faced like processing power, chip area, convergence rate and cost of the chip. These problems present a need for a portable, low power, area efficient hardware accelerator to accelerate the process of such learning.

This problem is targeted by implementing a hardware schematic architecture for Q-learning using Artificial Neural networks. This architecture exploits the massive parallelism provided by neural network with a dedicated fine grain parallelism provided by a Field Programmable Gate Array (FPGA) thereby processing the Q values at a high throughput. Mars exploration rovers currently use Xilinx-Space-grade FPGA devices for image processing, pyrotechnic operation control and obstacle avoidance. The hardware resource consumption for the architecture has been synthesized considering Xilinx Virtex7 FPGA as the target device.

ACKNOWLEDGMENTS

I would like to express my profound gratitude to my Chair and Mentor Dr. Jekan Thanga for presenting me with such an opportunity, for his guidance, support and motivation. I would like to specially thank him for introducing me to the field of Artificial Intelligence.

I would like to express my sincere gratitude to Dr. Fengbo Ren and Dr. Jae-sun Seo, for taking their valuable time out and agreeing to be part of my defense committee. Also, I would like to thank Dr. Fengbo Ren for providing me with an access and license to Simulation tools.

I appreciate the support of my colleagues at SpaceTREx Lab, and would like to thank graduate advisors Lynn Pratte, Toni Mengret and Sno Kleespies for their timely help.

Finally, I would like to thank my family for their immense support and encouragement throughout my master's studies at ASU.

TABLE OF CONTENTS

Page

CHAPTER Page

LIST OF TABLES

LIST OF FIGURES

Figure Page

Figure Page

CHAPTER 1

INTRODUCTION

1.1 Background

Space missions are often extremely challenging with long communication latencies and operate in a complex environment compelling a need for implementing autonomous and Artificial Intelligence (AI) algorithms on board. Typically, human level artificial intelligent algorithms are computationally intensive and require huge processing power, making it a concern for running such algorithms on space rovers [2]. The current maximum utilization of AI present in the Mars Science Laboratory (MSL) rover is the AEGIS (Autonomous Exploration for Gathering Increased Science) software, which helps the rover to autonomously choose targets for the laser [3], as shown in figure 1.

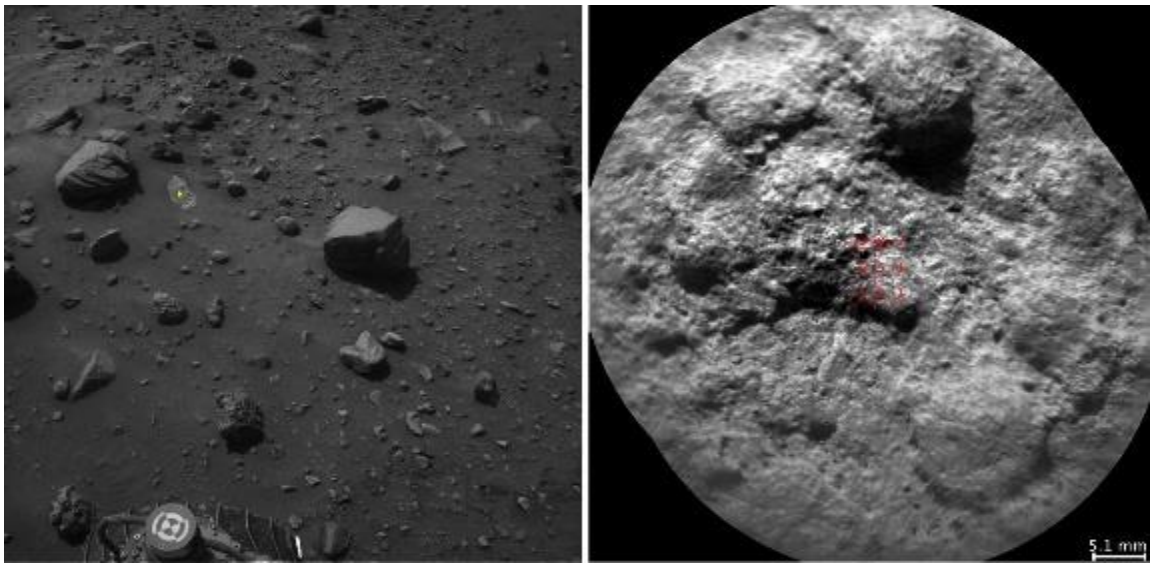

Figure 1. Target selection and Pointing Based on Navcam Imagery on MSL Rover [3].

Reinforcement learning [4], is one of the potential learning algorithms concerned on how an agent in a random un-modelled environment ought to take actions to maximize

the cumulative future reward. Through trial-and-error interactions with its environment, RL aids a robot to autonomously discover an optimal behavior. In RL, the human interaction with the robot is confined to providing feedback in terms of an objective function that measures the one-step performance of the robot rather than detailing generalized close-form solution [5]. This is one of the key advantages of RL because in real-world applications there is seldom any chance of finding a closed form solution to the problem. The objective in RL is specified by the reward function, which either acts as reinforcement or punishment conditioned on the performance of the autonomous agent with respect to the desired goal.

Reinforcement learning has been in existence for 30 years, however the most recent innovations in combining reinforcement learning with deep neural networks [6][7][8][17] has paved a strong way for achieving human level intelligence. The system with such approach has proven to beat humans in various video games by only using the pixels of frames and game scores [7]. Some of the examples in robotic applications include, robots learning how to perform complicated manipulation tasks, like the terrain adaptation of locomotion skills using deep reinforcement leaning [8] shown in figure 2, and Berkley robot, stacking Legos [9]. Robotic platforms running such algorithms have huge computational demand with often unrealistic execution times and huge processing power when implemented using traditional Micro controllers or CPU's [10]. This requirement paved way for development of accelerators which in theory can speed up the computations at low-power. These accelerators consist of either Graphic Processing Units (GPU's), Application Specific Integrated Circuits (ASIC's) or FPGA's for delivering such a huge performance requirement.

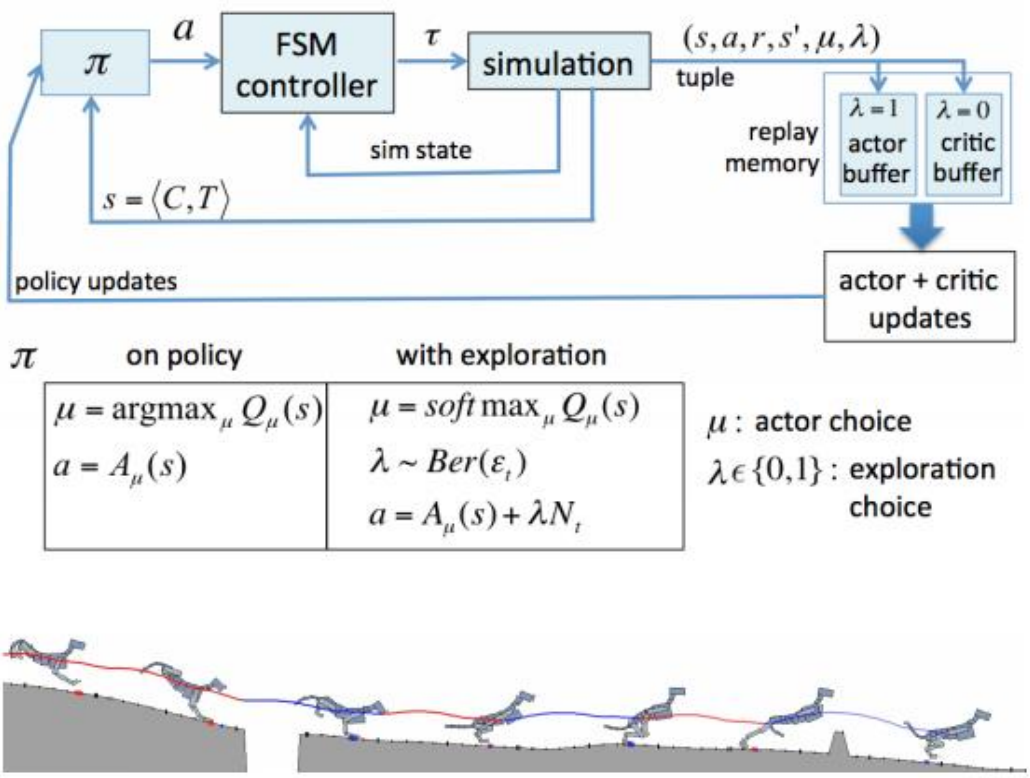

Figure 2. Deep Reinforcement Controller for Terrain Adaptation [8].

Over past few years there had been a major prevalence in cloud based computations which use deep learning algorithms on a cluster of high processing compute units. Recently google developed an artificial brain which learns to find cat videos, using a neural network that is built on 16,000 compute processors with more than one billion connections [11]. Such huge servers often consume a lot of power, for example one large, 50,000 square feet data center consumes around 5MW of power which is enough to provide electricity for 5,000 houses [12]. Power and radiation is a major concern for space exploration rovers. The curiosity rover's processor RAD750, a radiation-hardened with a PowerPC architecture running at 133MHZ having a throughput of 240MIPS, consumes a power of

5W, while the Nvidia Tegra K1 mobile processors specifically designed for low power computing with 192 Nvidia CUDA Cores consumes 12 W of power.

FPGA's are starting to emerge as a reasonable competition for GPU's in implementing deep learning algorithms [13][14][15]. A timeline of FPGA development and its applications in neural networks is shown in figure 3. Microsoft's research paper on accelerating deep convolution neural networks using specialized hardware demonstrates its 2x performance improvements in accelerating Bing ranking [16]. Due to its prevalence and existing wonders of combining deep neural networks with Q-learning, an accelerator architecture for Q-learning using neural networks is implemented to thoroughly exploit its performance.

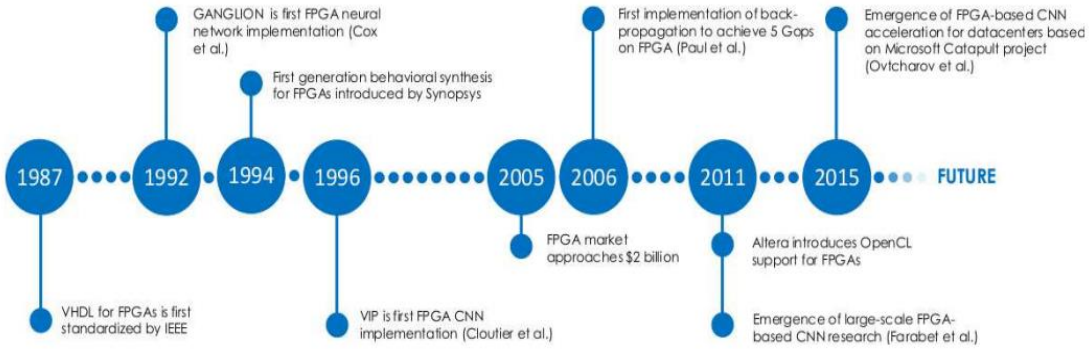

Figure 3. Timeline of FPGA Based Neural Networks as of 2015 [21].

1.2 Problem Statement

Q-learning algorithm using neural networks tend to converge slower and therefore have a slower completion time when compared against traditional supervised learning approach. The reason can be attributed to the need for an exhaustive exploration of the environment. Autonomous robotic systems running such algorithms tend to be slow in performance and energy inefficient when implemented on a traditional microcontroller. An

FPGA based fine grained parallel architecture of the algorithm exploits the parallelism in a neural network, thereby processing Q values with higher throughput. This approach reduces the learning time for Q value.

1.3 Objective

The objective of this thesis is to come up with an implementation of a hardware accelerator architecture for Q-learning using perceptron and extend it to an architecture design for Q-learning using Multilayer Perceptron. The work includes demonstration of throughput calculation for floating point and fixed point architectural implementation when executed on a simple and complex environment. The performance is compared against CPU based implementation of the algorithm. Total on chip power consumption based on the hardware resource utilization is also demonstrated for each of the architectures.

1.4 Thesis Structure

The structure of thesis is as follows, In Chapter 2 the concepts of Reinforcement learning, Q-learning and Artificial intelligence accelerators have been discussed. Chapter 3 demonstrates the state space description and working of a Q-learning simulated system on a CPU followed by the accelerator architecture for the Q-learning using a single perceptron and Multilayer perceptron. Chapter 4 discusses on the results obtained and Chapter 5 draws conclusions and presents a discussion about the future work.

CHAPTER 2

LITERATURE REVIEW

2.1 Machine Learning Algorithms

Machine learning algorithms aim to solve the learning problem for an Artificial intelligent system, by making a machine perform a function without explicitly mentioning what the function is. It is a field of research, aiming to design machines that improve their performance by adapting with the environment. Machine learning algorithms are broadly classified into 3 types based on the type of problem a machine deals with and how a machine interacts with its environment or experiences. Following are the 3 major types of machine learning algorithms.

1. Supervised Learning Algorithm
2. Reinforcement Learning Algorithm
3. Unsupervised Learning Algorithm.

Using supervised learning algorithm, a machine models itself based on the predefined training set. The training set includes the input data and the expected data also called as a labelled data. The performance of the supervised learning algorithm is based on how well is the training set designed, and how efficiently does the error help in learning process.

Reinforcement learning algorithm is one step ahead of supervised learning which uses a real-time reinforcement mechanism instead of training targets to model the system. For every decision the machine takes, rewards are obtained and are applied as a feedback to the machine. The aim of the machine is to maximize the total sum of rewards over the course of its actions.

Unsupervised learning algorithm, uses unstructured data with no labels. The goal of the machine running an unsupervised learning algorithm is to describe a structure of data based on how the data is organized. Complex input data is converted into clusters of organized and simple structures.

Reinforcement learning and one of its variants Q-learning is discussed in detail, in the following sections.

2.1.1 Reinforcement Learning Algorithm

Reinforcement learning, is a powerful machine learning algorithm that combines the concepts of dynamic programming [20] with supervised learning trying to solve human level problems. The algorithm is concerned on how an agent in a random un-modelled environment, ought to take actions so as to maximize the cumulative reward.

The traditional way of formalizing a reinforcement learning algorithm is to represent it as a Markov Decision Process(MDP), in which transition from one state to another state is only dependent on the current state and the action the agent takes. The decision is independent of the future states. Following are the simple sequence of steps an agent running reinforcement learning algorithms takes into consideration.

1. At each single step, an agent executes an action from a set of possible actions based on a policy which is called as an action selection policy.
2. The environment detects the action performed by the agent and emits an observation forcing the agent to move to a new state. In general, for robotic environments the state movement is stochastic.

3. The agent then eventually maps the observation with its reward function and generates a reward signal which acts as reinforcements. The reinforcements can either be positive or negative.
4. Based on the reward generated and the state action pair, the agent makes modifications in the utilities of selecting an action when present in a state and also updates its action selection policy.

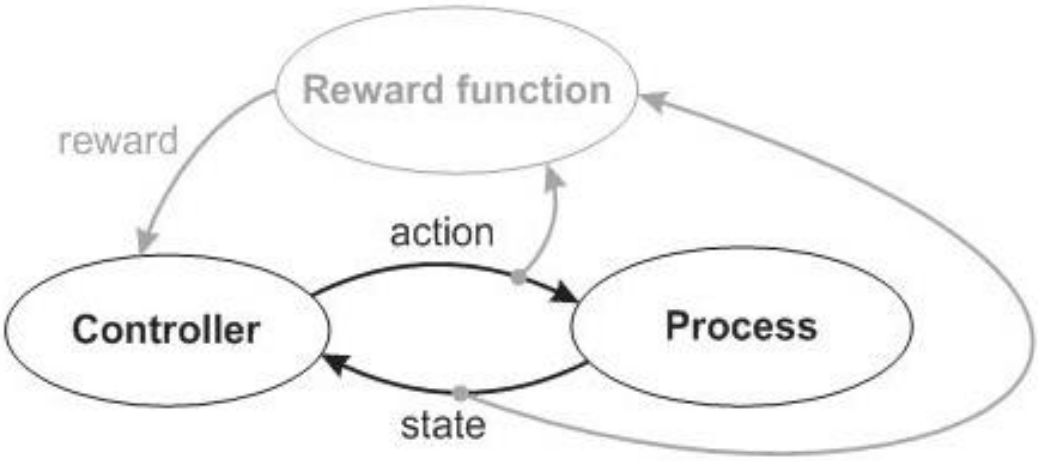

Figure 4. Reinforcement Learning Demonstration with Reward Depicted in Gray [22].

The concept of exploration and exploitation plays a major role in Reinforcement learning algorithm. With an exploitation policy, an agent only tries to perform those actions which result in maximum immediate rewards, not taking into account the future rewards. However, by an exploration algorithm, the agent does not consider the immediate rewards, but always try to look for the future rewards increasing the time for convergence.

Discounting future rewards, provides an intermittent solution for the exploration and exploitation problem. According to discount future reward policy, the total reward

obtained in any state is a combination of immediate reward and the discount factor of future rewards. Assuming that the Markov Decision Process has n states, where by the end of nth state, the agent has successfully performed all of its optimal actions. At any point 'τ' the total future reward 'R_τ' is equal to the sum of immediate reward r_τ and the discounting factor (γ) times the total future reward $R_{\tau+1}$. Figure 5 demonstrates the concept of discounting future reward for a robot traversal. The total reward at a time step τ, R_τ can be expressed as in equation (1), where $R_{\tau+1}$ is the overall future rewards.

$$R_\tau = r_\tau + \gamma R_{\tau+1} \quad (1)$$

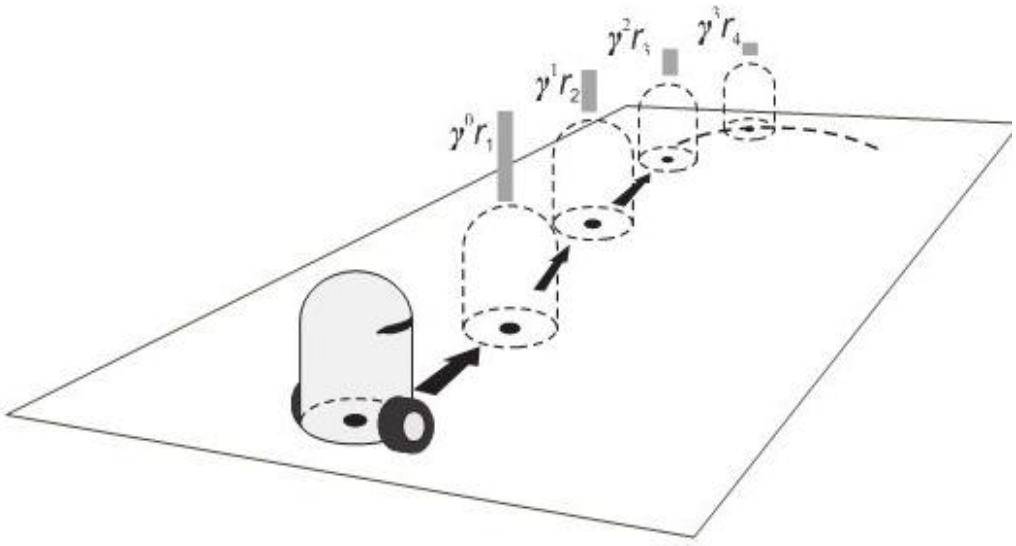

Figure 5. Discounting Rewards Along the Path of the Robot [22].

Reinforcement learning agents are classified into 2 types: Utility learning agents and Q-learning agents. Utility learning agent learns the utility function and selects action which maximizes the expected utility, while the Q-learning agent learns the action-utility function, that is the expected utility of taking an action 'a' in a state 's'. The utility learning

needs to know the model of the environment beforehand so as to determine the utilities. These utilizes are determined only by knowing the state to which performing an action leads to. However, Q-learning doesn't need to know the model of the environment as it has a capability of comparing utilities of its choices of actions without having the knowledge of the upcoming state. Q-learning in more detail is discussed in the next section followed by the introduction of another similar learning algorithm called State-Action-Reward-State-Action (SARSA) Algorithm.

2.1.1.1 Q-learning Algorithm

Q-learning is one of the reinforcement learning techniques which finds and selects an optimal action based on an action selection function also called as a Q function [1]. The Q function helps in providing the utility of selecting an action, which takes into fact that optimal action 'a' selected in a state 's' leads to a maximum discounted future reward R_{t+1} as in equation (1), however only if we continue to select optimal actions from that point on. The Q function can be represented as follows.

$$Q(s_t, a_t) = maximum(R_{t+1}) \tag{2}$$

The future rewards are obtained through constant iterations selecting one of the actions based on existing Q values, performing an action in the future and updating the Q function in current state. Assuming that an optimal action in future based on the maximum Q value has been picked (However, picking an action based on maximum Q value might not always be the case). The equation (3) demonstrates the action policy selected based on the arguments of the maximum Q value. More details on action selection policy is discussed later in this section.

$$\pi(s) = arguments\ (maximum\ Q_t(s, a)) \quad (3)$$

After selecting an optimal action, the agent moves to new state, obtaining rewards during this process. Assuming the new state to be s_{t+1}, the optimal value for next state is found out by iterating through all the Q values in next state for various actions a' and finding an action a'_{t+1} which produces an optimal Q value in that state.

$$Q_o(t+1) = max_{a'} Q(s_{t+1}, a'_{t+1}) \quad (4)$$

Based on the optimal Q value of future state, the Q value for the present state is updated using equation (5).

$$Q(s, a) = Q(s, a) + \propto [r + \gamma . Q_o(t+1) - Q(s, a)] \quad (5)$$

The equation (5) is the basis of Q-learning algorithm, described in [1]. The constant parameters of the equation are $\propto$ and $\boldsymbol{\gamma}$. $\propto$ is the learning rate or learning factor with a value between 0 and 1, the learning factor determines how much a Q error update influences the current Q values. Large learning factor overshoots the local minimum of the Q function. With learning rate equal to 1, the Q values cancel out, resulting in equation (6). This represents a direct update of Q value with its target value. And with learning rate equal to 0, the value of Q never updates and remains same as before thereby stopping learning from happening.

$$Q(s, a) = r + \gamma . optimal(t+1) \quad (6)$$

Another constant, $\boldsymbol{\gamma}$ is the discounting factor also set between 0 and 1. The discounting factor takes into fact, what percentage of future rewards has to be taken into consideration relative to the immediate reward. At discounting factor equals to 0, the agent only takes immediate rewards into its consideration not considering the future rewards.

This leads to the concept of exploration and exploitation tradeoff as discussed before. Pseudo code for Q-learning is shown in figure 6.

```
funciton QLL
    Initialize Q(s,a) randomly
        for each iteration
            random Initialize(s)
            for each step of iteration
                Select an action form all possible actoins
                using on of the action selection policy
                Observe reward, nextState
                Use equation (5) to update the Q value
                Move to the next state s(t+1)
            until final state reached
        until end of iteration
```

Figure 6. Pseudo Code of Q-learning Algorithm

The above discussions, considers choosing an action greedily, that is choose an action with its maximum Q value in the next state. However, there are other action selection policies which are implemented in real-time.

a. Random action selection policy: An action is chosen at random irrespective of the Q value, such selection policy can be useful for an agent that is present in a complex environment trying to initially understand the environment.
b. Greedy Action policy: This action selection policy is used to select an action greedily, which in the case of Q-learning, is to select an action with maximum Q value.
c. ε- greedy Action policy: The action selection policy is used to select a greedy action with a probability of ε.
d. Boltzmann Action policy: This is the most common selection policy, which chooses the estimated best with a probability proportional to $e^{Q(x,a)/T}$.

There are many implementations of Q-learning algorithm in real-time systems [33][6][34]. Most of the them are based on 2 types of approaches: Q-learning using Q table and Q-learning using neural networks. Q-learning using Q table, stores the Q values in runtime in a 2D storage space. Table1, demonstrates how a Q table looks like, the Q values for each of the state-action pair are stored in the Q table.

The Q table updates the Q values using equation (5). The size of Q table is equal to the number of actions multiplied by the number of states, which acts as a bottleneck for an environment with a complex state space and with multiple number of possible actions.

Q- Table	Actions[1]	Actions[i][j]	Actions[n]
State[1]	Q[1][1]	Q[1][j]	Q[1][n]
State[i]	Q[i][1]	Q[i][j]	Q[i][n]
State[n]	Q[n][1]	Q[n][j]	Q[n][n]

Table 1. Sample Q Table Implementation

Q-learning with neural networks, eliminates the usage of Q table with neural network acting as a Q function solver. Instead of storing all the possible Q values, they are estimated based on the output of a neural network. The neural network is trained with Q value errors obtained from equation (5).

The next section discusses about the SARSA algorithm and is followed by artificial neural networks, and equations pertaining to the training of neural networks. These equations are the basis for the neural network architecture design of the hardware accelerator.

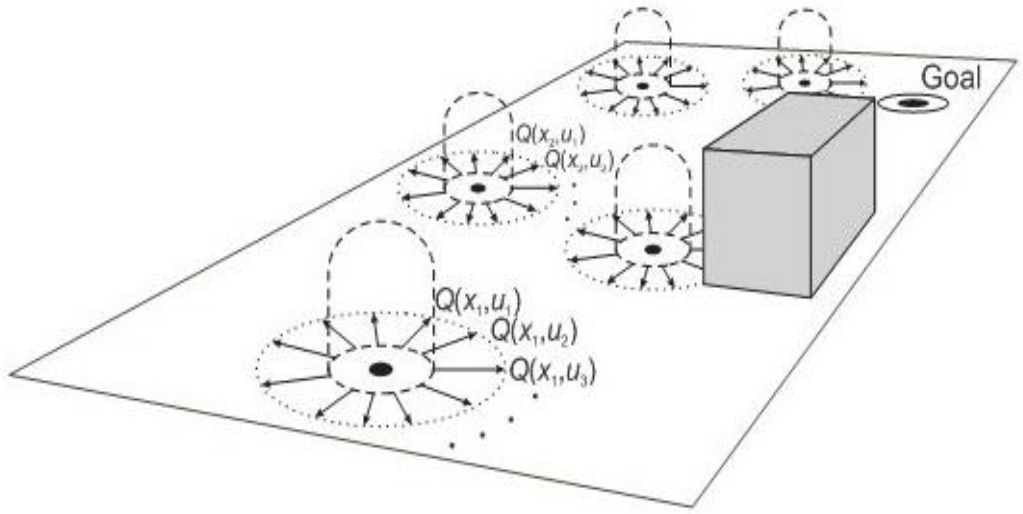

Figure 7. Storing Q values for each State Action Pair [22]

2.1.1.2 State-Action-Reward-State-Action (SARSA) Algorithm

State-Action-Reward-State-Action (SARSA) algorithm is another form of reinforcement learning algorithm which takes into consideration, the usage of control policy to update the action selection pair. The algorithm differs from Q-learning in fact that, Q-learning is considered to be an on-policy learning algorithm while SARSA is considered to be an off-policy learning algorithm. The SARSA learning algorithm is shown in figure 8.

Q-learning provides a better set of Q-values in a frequently changing policy when compared with SARSA due to the fact that Q-learning looks into the future while SARSA updates its Q-values only with immediate rewards. SARSA algorithm is used to learn a non-optimal but safe policy which avoids the negative or low reinforcements not considering if the path is optimal. Performance comparison of SARSA and Q-learning algorithm for a maze problem is presented in figure 9.

Initialize $Q(s, a) \leftarrow 0$ and $e(s, a) \leftarrow 0$ 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 (ε-greedy)

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

    For b in Action Set:

      $e(s, b) \leftarrow 0$

    $e(s, a) \leftarrow 1$

    For all s in State Set, a in Action set:

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

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

    $s \leftarrow s'$

    $a \leftarrow a'$

  until s is terminal

Figure 8. Pseudo code for SARSA Learning algorithm [35]

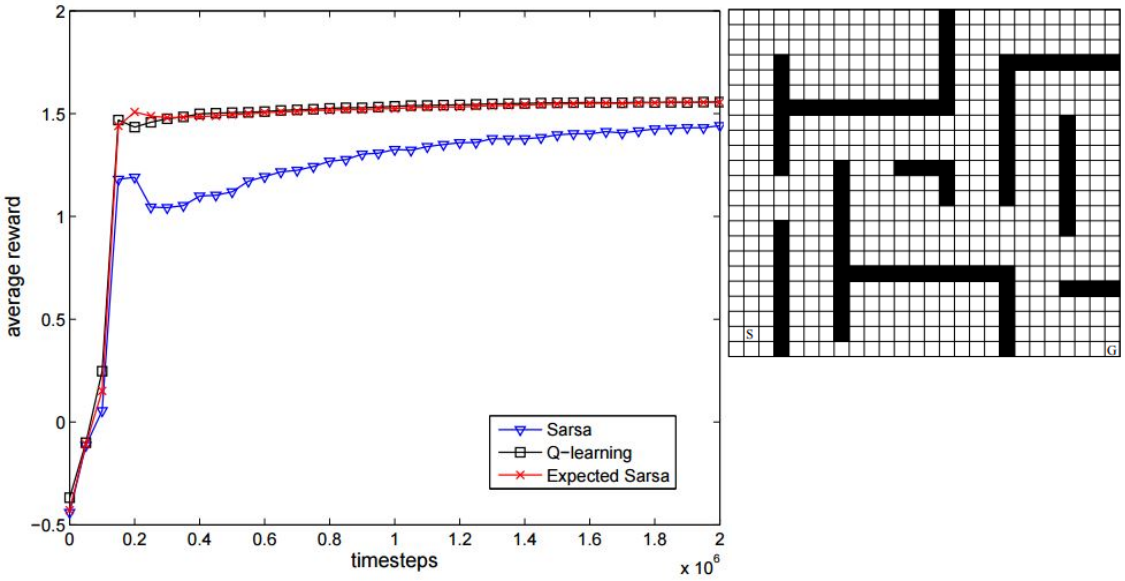

Figure 9. Performance comparison of reinforcement learning algorithms on the maze problem [36]

2.2 Artificial Neural Networks

Artificial neural networks (ANN), is a field of machine learning inspired by biological neurons. A neural network is made up of simple but many nerve cells also called as processing elements. Multivariable dependency problems cannot be formulated using an algorithm. There is need for a learning based approach for solving such problems. Neural networks with the capability of learning as its most significant aspect is used for such learning process. Classical network structures like the perceptron and multilayer perceptron with their learning procedures are demonstrated in further sections.

2.2.1 Perceptron

Perceptron (modelled as a biological neuron) is as a single layer neural network, with multi-dimensional inputs and a single output. A perceptron constitutes weights for each of the inputs and a single bias, as shown in figure 10. The output of a perceptron is calculated using equations (7-8). The equations (7-17) are derived from [19].

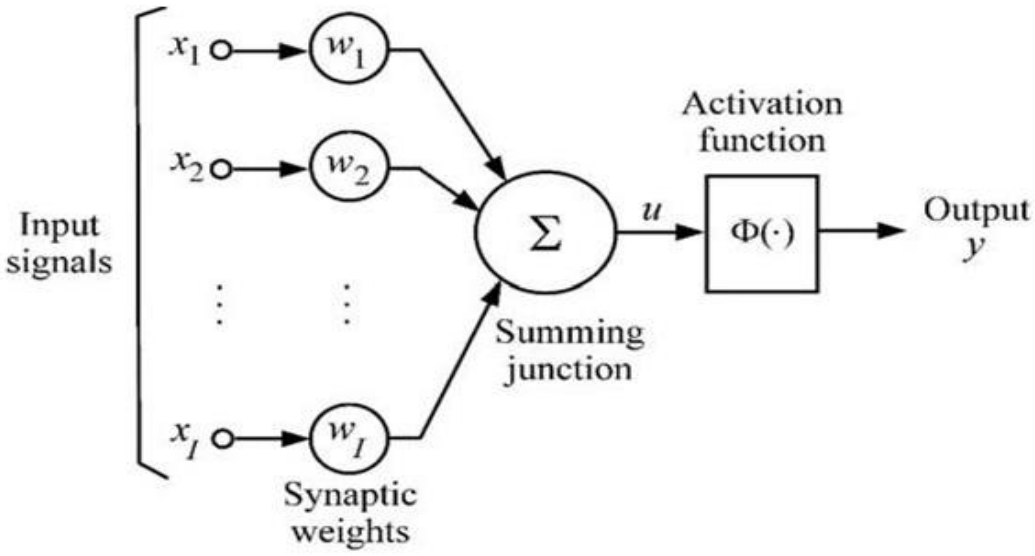

Figure 10. Schematic of a Perceptron [19]

The weighted sum of all the inputs 'net' is calculated using the equation (7) where N is the number of inputs to the perceptron. The summation of weighted inputs is termed as Multiplier and accumulator (MAC) which would be frequently used in the design for hardware schematic.

$$net = (\sum_{i \in N} x_i * w_i) \tag{7}$$

The output of a perceptron, also called as the firing rate, is calculated using equation (8), where f is the activation function. There are many activation functions implemented from which sigmoid function has been chosen due its bounding, easy differentiability and monotonic property [23] which eases the training process in a simple network.

$$O = f(net) = \frac{1}{1+e^{-net}} \tag{8}$$

Equations (7-8) constitute the feed forward algorithm for single perceptron. Backpropagation algorithm is used to update the weights (w_i) of neural network. The error that needs to be back propagated is calculated based on the expected output, which in case of supervised learning agent is obtained as an input from the user during training process. The RL algorithm obtains the expected output using equation (5). The error value is calculated using equation (9), where $f'(net)$ is the derivative of activation function.

$$\delta = f'(net) * (T - O) \tag{9}$$

The error thus obtained is propagated backwards and weights are updated using equations (10-11), where C is the learning factor.

$$\Delta W = (C * O * \delta) \tag{10}$$

$$W_{new} = Wprev + \Delta W \tag{11}$$

An extension to the perceptron, Multilayer perceptron learning is discussed in next section.

2.2.2 Multi-Layer Perceptron

Multilayer perceptron (MLP) is a neural network model with more than one layer of neurons or processing elements. The output of a multilayer perceptron is the function of outputs of each of the individual processing node and the weights of links between each of the nodes. The weighted sum of the MLP is calculated using equation (12).

$$net_i = (\sum_{j \in A} O_j * W_{ji}) \quad \forall i , i \in B \tag{12}$$

B and A in equation (12) represents current and previous layer neurons. The output of the MLP is calculated using equation (13), where f is the activation function.

$$O_i = f(net_i) = \frac{1}{1+e^{-net_i}} \tag{13}$$

Error value for the MLP is calculated using equation (14). The error value is based on overall firing rate of MLP which is extracted from the last layer of neural network. C in the equation (14) denotes the last layer neurons and f'(net) is the derivative of sigmoid function.

$$\delta_i = f'(net_i)(T_i - O_i) \forall i , i \in C \tag{14}$$

The error δ_i, from the output layer is propagated to the hidden and input layers using equation (15), D and E in equation (15) represents non output layer and output layer neurons respectively.

$$\delta_i = f'(net_i) * (\sum_{j \in E} \delta_j * W_{ij}) \forall i , i \in D \tag{15}$$

The change in weights are calculated and updated using equations (16-17) respectively, in which C is the learning rate, while A and B represents current and next layer neurons.

$$\Delta W_{ij} = (C * O_i * \delta_j) \forall i, j ; i \in A ; j \in B \tag{16}$$

$$W_{ij} = W_{ij(prev)} + \Delta W_{ij} \quad (17)$$

Multilayer perceptron is one of the variants of Deep Neural Networks. Any ANN with more than one hidden layer is termed as a deep neural network. One of the problems faced by a multilayer perceptron with more than one hidden layer is the vanishing gradients problem [24]. The error while propagating backwards as represented by equation (15), gets smaller and eventually vanishes. A solution presented for such a problem in a deep multilayer perceptron neural network is to use a single layer auto encoder to determine the parameters initially for each of the layers [25]. The main advantage of the deep learning is the usage of a large amount of data to initially craft the features in an unsupervised manner.

In the next section the types of parallelism neural network presents are exploited.

2.2.3 Types of Parallelism in Neural Networks

Neural networks have an advantage in terms of exhibiting various levels of parallelism. The types of parallelism [19] in neural networks are as follows.

1. Node Parallelism: Each individual neuron can be implemented with individual resources for each of the neuron. This type of parallelism can typically be implemented on a Field Programmable Gate Array (FPGA), due to the potential existence of large number of FPGA cells.
2. Layer Parallelism: Multilayer perceptron architecture presented in section 2.2.2 consists of more than one layer which can be parallelized with each of the layer having resources of its own.

3. Weight Parallelism: A single neuron as mentioned in section 2.2.1 consists of blocks of multiple multipliers and an accumulator, which can be parallelized with individual combination of weights and inputs having their own computational units.

The following section demonstrates the concept of implementing the Q-learning algorithm with neural network.

2.3 Q-learning using Artificial Neural Networks

At the end of section 2.1.1.1 the problem faced by the Q table approach has been discussed, and also the advantages of using neural networks for Q-learning has been mentioned. Section 2.2.1 and Section 2.2.2 discussed supervised learning using neural networks in which the Target (T) values are predefined and are given as an input to the system as a part of the training process, however the Q-learning and other reinforcement learning techniques rather than using the predefined target values, use the estimate of errors based on the future values to perform the learning process.

In a neural network, Q values for a given state-action pair is stored in the form of weighs and biases. To find the Q value for a given state and action, a feed forward step is performed with inputs as the state and action vector and executing a feed forward algorithm using either of equations (12-13) or equations (7-8).

The learning process happens by performing the backpropagation algorithm. During the learning process, Q values for all possible actions in a state are calculated by executing the feed forward algorithm for multiple number of actions. An action is chosen by using one of the action selection policy, thereby a new state is determined based on the action chosen. Q values for each of the new state are calculated for all possible future

actions. This is followed by the error generation step which uses equation (6) to calculate the error values. The weights are updated based on the error calculated using equations (9-11) or equations (14-17).

2.4 Hardware Accelerators

Hardware accelerators are the functional blocks which are optimized and are designed to offload computationally intensive tasks from the CPU or in fact any microcontroller. These accelerators work alongside CPU's in-order to increase the performance of the system and improve the energy efficiency of the system. System scheduler inside a CPU can be used to schedule different applications among different accelerators present in a heterogeneous system. There are many forms of hardware accelerators like Graphic Processing Units (GPU), Digital Signal Processors (DSP), Application specific integrated circuits (ASIC), and Field Programmable gate arrays (FPGA's). GPU's generally compute applications with a high throughput, however they consume a huge amount of power. Nvidia's Tesla K-40 consumes as much as 235W of power when running double precision matrix-matrix multiplication (DGEMM) [26]. FPGA's offer a compelling choice with a high throughput per power when compared with GPU's (Figure. 11) and short development and low cost overhead when compared with ASIC's.

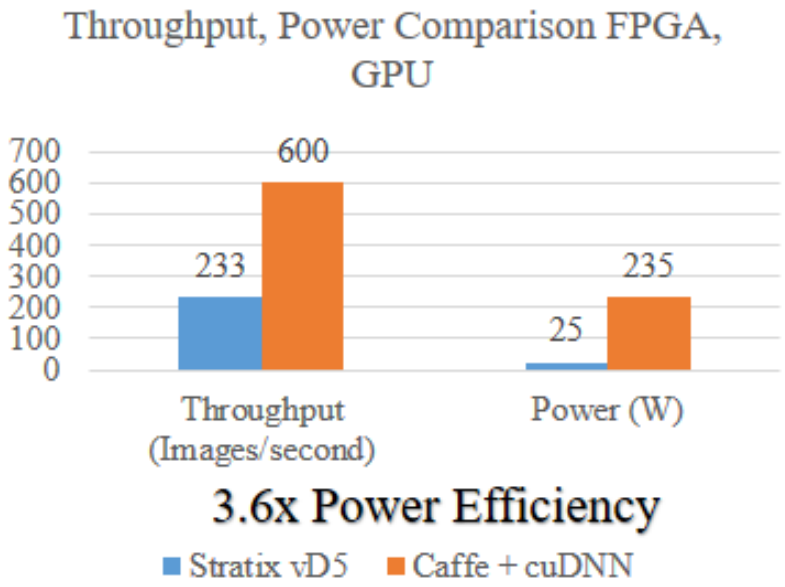

Figure 11. Power Efficiency Comparisons for Convolution Neural Networks [22]

2.4.1 Artificial Intelligence Accelerators

In past couple of years, we see an emerging class of microprocessor design like the Vision Processing Units [27], Tensor Processing Units [28], Nvidia DGX-1 [29], Zeroth Neural Processing Unit (NPU) [30], etc. which are specifically designed to offload Artifical Intelligence applications like the computer vision, deep learning, and other data intensive algorithms. These accelerators have their applications in robotics, self-driving cars, speech recognition, search engines, voice control, etc. ASIC, General Purpose Graphic Processing Units (GPGPU'S) and FPGA'S are the widely used AI accelerators.

2.4.2 Field Programmable Gate Array Overview

For past 20 years, there has been a consistent effort to introduce a greater programmability into digital devices. Considering the perspectives in hardware domain, there is always a drive to design devices with low cost, low power, smaller size and with

high performance which presents a need to design a custom integrated circuits also called as Application Specific Integrated Circuits (ASIC). However, for large productive runs, the custom design based approach is not cost efficient. On the end of a software engineer perspective, software applications can be developed onto standard processor architecture like PowerPC, ARM, Intel, etc. Though such implementations are faster and have a very low cost overhead, the non-existing direct relationship between hardware and software creates a performance drop. Hence to solve such a problem, Field Programmable Gate Arrays (FPGA's) have been developed.

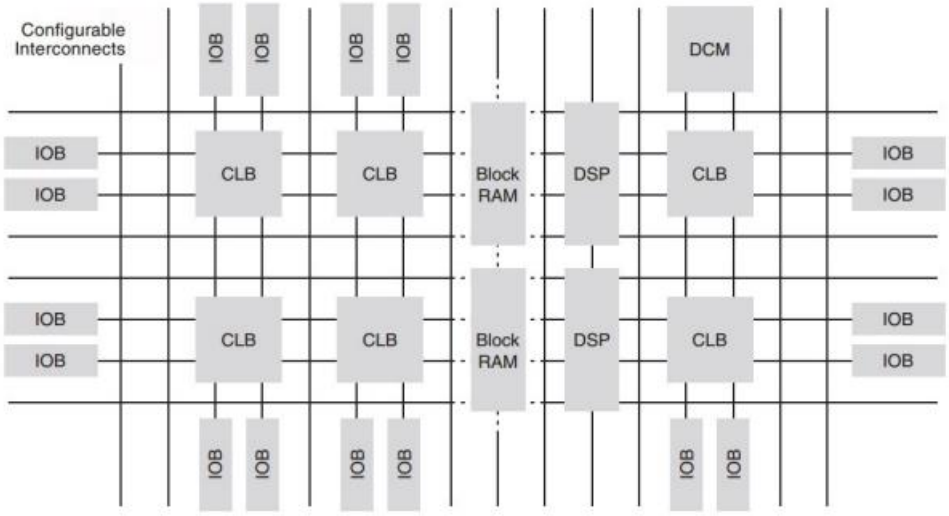

Figure 12. Reconfigurable Devices on a Xilinx FPGA [31]

Field Programmable Gate Array (FPGA) got its name because of the very less overhead time required to program it on field. FPGA's are made up of reconfigurable set of resources, which are configured based on the type of function being implemented on it. They are made up resources like Configurable Logic Blocks (CLB), Block Random Access Memories (BRAM), Digital Clock Managers (DCM), Digital Signal Processors (DSP) and Input-output blocks (IOB) as shown in figure 12. Pipeline stages, memory hierarchy,

operators in data path and interconnects are customized for specific application running on the FPGA.

CLB's are the building blocks of FPGA which perform user-specified logic functions. The array of CLB's in FPGA are arranged in columns and rows, with routing channels acting as an interconnect between them. The structure of a FPGA CLB is shown in figure 13.

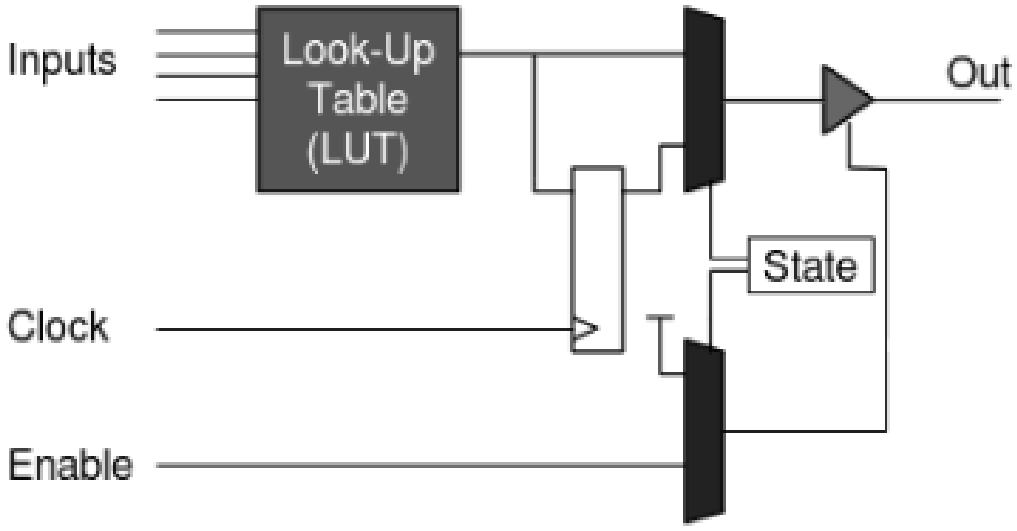

Figure 13. Schematic of FPGA Configurable Logic Block [32]

CHAPTER 3

METHODOLOGY

This chapter presents the implementation methodology of the Q-learning algorithm. A real-time implementation of the algorithm has been simulated on a small state space rover. It is followed with demonstrations of hardware accelerator architecture implementation for Q-learning.

3.1 Design Goals

For demonstrating Q-learning algorithm on real-time rover, 4 main factors are to be taken into consideration.

1. The goal of the rover, indicating the variable the rover aims to maximize.
2. The number of sensors and their degrees of freedom, so has to clearly estimate the state-space of the rover.
3. The possible actions a rover can perform, which gives the estimation of the size of the action space.
4. The reward mechanism, indicating the value of rewards the rover acquires during its process of learning.

3.2 System Description and Execution

Q-learning algorithm is simulated on the CPU with the help of Mobile robot simulator which features a graphical interface, for simulating robots and objects. The movement of the robot is configured with a BASIC editor. The rover is simulated with 2 wheels, 2 proximity sensors and 2 temperature sensors. The aim of the rover is to reach the point of high temperature, while avoiding obstacles. The rover is simulated for multiple start positions.

3.2.1 Geometric Description

The bot has been configured with the platform and wheel parameters as in table 2. The parameters have been chosen in a way so as to reduce the complexity of sharp turns and to create a satisfactory temperature difference between 2 sensors.

D	Platform Diameter	0.5	m
d	Distance between wheels	0.35	m
D_w	Wheel Diameter	0.2	m
W_w	Wheels Width	0.04	m

Table 2. Platform and Wheel Parameters for the Simulated Bot

The temperature sensor cone of 40^0 has been chosen as shown in figure 14, to cover the complete forward path not leaving any undetected holes in front of the rover. Proximity sensors with a range of 0.3m – 2m has been considered based on the platform diameter value. The value is in between a range of 1 to 4 times the platform diameter.

Sensor cone	40	^{o}
Proximity sensors	0.3 – 2	m
Percentage of misreading	0.2	%

Table 3. Sensor Parameters for the Simulated Bot

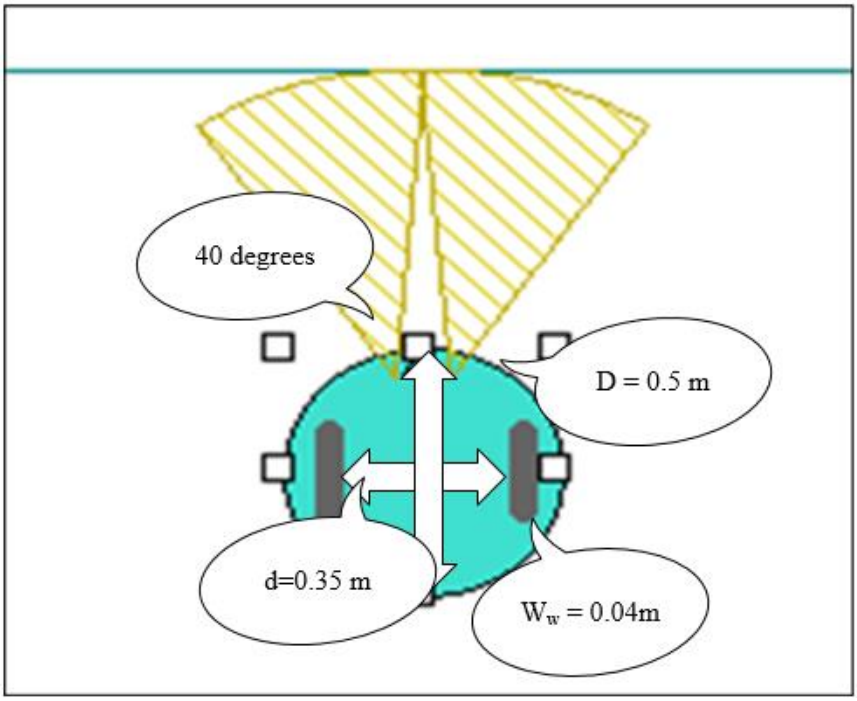

Figure 14. Simulated Bot with Temperature Cone and Geometric Parameters

3.2.2 State Space

The state space of the bot is formulated based on the sensor readings, which are dependent on the environment the bot is currently in. The environment is modelled in the simulator as shown in figure 15. Let the value on the left sensor be L_T, right R_T, Front F_P and back B_p. The possible states for the 2 temperature sensors are 3 in total, when left temperature sensor value greater than right ($L_T > R_T$), when right sensor value is greater than left ($L_T < R_T$) and if both are equal ($L_T == R_T$). There are 2 additional states for temperature sensors, to make sure that the bot does not stop pursuing its goal which is determined based on change in temperature for 3 consecutive iterations. This is indicated by G_T and G_F representing when there is a change, and when this no change respectively. Similarly, there are 2 states for front proximity sensor indicating if there is an obstacle in

front (F_{PT}) or not (F_{PF}) and 2 states for proximity sensor at the back indicating if there is an obstacle in back (B_{PT}) or not (B_{PF}).

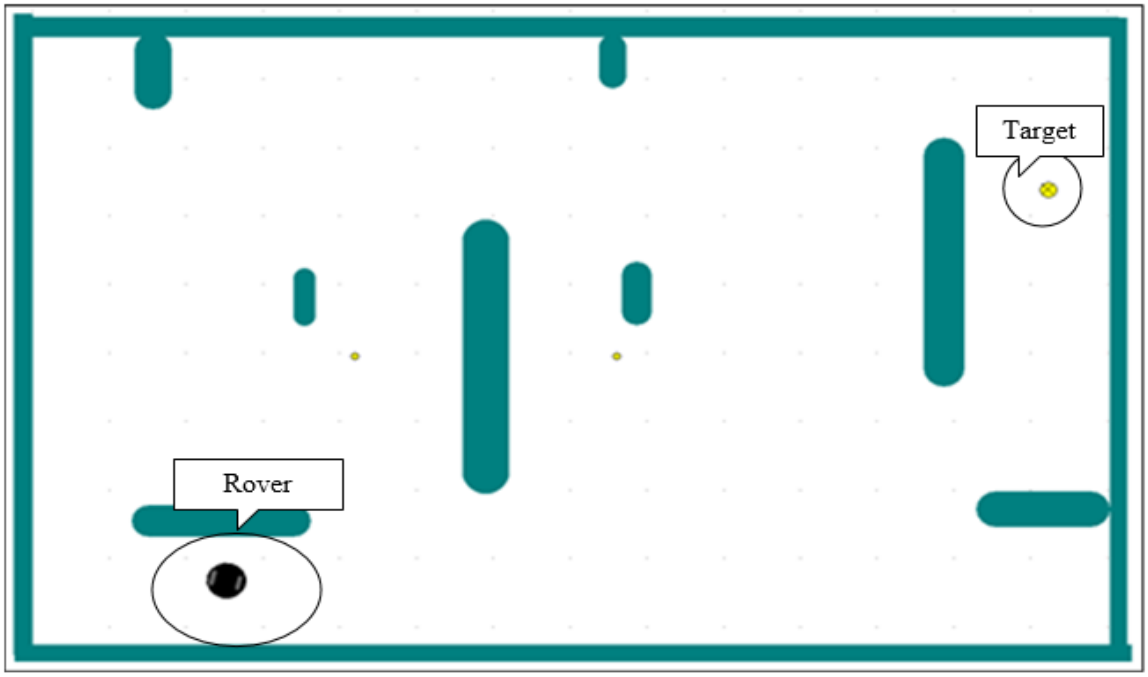

Figure 15. One of the Environment Simulations

The total possible states come out to be 24 in total. However, the realistic state space when considering image pixels as the input would be exponentially large when compared to our simulated state space. For example, for the implementation of Q-learning in ATARI, the number of states are 10^{67970}.

3.2.3 Action Space

Possible actions a rover can take in this situation are to either move front, back, left, right or stay at the same place. This has been modelled by the movement of individual wheels. Each wheel can have 3 possible actions, stop (0), move forward (1), or move

backward (-1). Combining actions for both the wheels, the action space has a size of 9(3*3) possible actions (A0-A8). Combining state space and action space, the Q table looks in the format as shown in table 4.

	A0	A1	A2	A3	A4	A5	A6	A7	A8
State 0	Q00	Q01	Q values						Q08
State 1	Q10	Q11							Q18
State 2	Q20	Q21							Q28
…… .									.....
State 22	Q220	Q221							Q228
State 23	Q230	Q00	Q232	Q233	Q234	Q235	Q236	Q237	Q238

Table 4. State Action Space for the Rover

3.2.4 Reward Mechanism

Rewards play a major role in the learning of bot. Reward mechanism has been chosen in a way as to avoid collisions and force the rover to reach its goal. Maximum reward is presented to the bot, when the bot reaches its goal, that is when a high temperature value is reached. Approaching a high temperature value is also a desired situation for which a sufficient positive reward is allotted. If there is no change in temperature values with its previous iteration, no rewards are allotted. A collision is to be highly avoided for which a negative reward is allotted if in case such a situation occurs. No prolong change in temperature is also undesired, hence a negative reward is allotted if the rover doesn't

change its state continuously for 3 cycles. Approaching or not approaching goal is determined based on the changes of temperature values, if the change is positive, it indicates the rover is approaching its goal else, the rover is not approaching.

State	Reward
Reached goal	10
Approaching goal	5
Not Approaching goal	-5
No change in readings	0
Collision	-10
Prolong no change	-10

Table 5. Reward Mechanism Simulated

3.2.5 Pseudo Code for Q-learning using Q table.

Q table representation of Q-learning algorithm consists of Q values stored in the form of Q table. Each table entry represents the Q value associated with the state-action pair. The size of the Q table depends on number of possible states an agent can be in and number of possible actions a rover can perform.

Q table is represented by a 2D array data structure, the array is indexed with state and action values. The sax basic language provides access to various inbuilt functions which are used to get the position of the bot, move the bot, set the direction and also set the speed for the bot.

```
Byte Bot                                        // Bot Representation
Double SensorTemp[2]                          // Temperature Sensor Value
Double SensorProx[2]                          //  Proximity Sensor value
Double df                              // Discount Factor
Double lr                              // Learning Rate
STATE0..STATE23 = 1..24

QLL_QTable(Q, s, a)
 Randomly Initialize Q[s][a]
 While Runs ≠ 0
        While FireReached() == False
                do RandomPlace()
                s ← GetState()
                a ← GetAction(Q, s)
                ActionReturn = PerformAction(a)
                if(ActionReturn ≠ 0) snew ← GetState()
                r ← GetReward(s, snew)
              UpdateQTable(Q, s, snew, r, a)
        endWhile
 endWhile
 end

UpdateQTable(Q, s, snew, r, a)
 foreach a in Q[snew][a]
          Qmax = Max(Q[snew][a])
 error = r + df*Qmax - Q[s][a] // error = reward + discountfactor)*(OptimalQnextstate)- Q(current)
 Q[s][a] = Q[s][a] + error          // Update the Q value
 End

GetState()
 if(sensorTemp[0] < sensorTemp[1])
        lcount = lcount + 1
        return [STATE0 - STATE7] ?  SensorProx[0], SensorProx[1], Lcount
 elseif(sensorTemp[0] > sensorTemp[1])
        rcount = rcount + 1
        return [STATE8 - STATE15] ?  SensorProx[0], SensorProx[1], Rcount
 elseif(sensorTemp[0] > sensorTemp[1])
        rcount = rcount + 1
        return [STATE8 - STATE15] ?  SensorProx[0], SensorProx[1], Rcount

GetReward(s, snew)
 return rewardTable[s][snew]
```

Figure 16. Pseudo code for Q-learning using Store Memory

3.2.6 Pseudo code for Q-learning using Artificial Neural Networks

Q-learning using Neural networks omits the usage of Q table, thereby reducing the space requirements for Q table storage. Initially a single perceptron with 6 inputs and one output, has been implemented to calculate the Q value. Each of the 6 inputs are associated with 6 weights, and an additional bias. Figure 17 shows the perceptron implementation; The target Q values are calculated according to equations (7-8) and the weights of the perceptron are updated using equations (10-11). The error generation process is not as direct as that of the supervised learning, there is need for more than one feed forward step to generate the error.

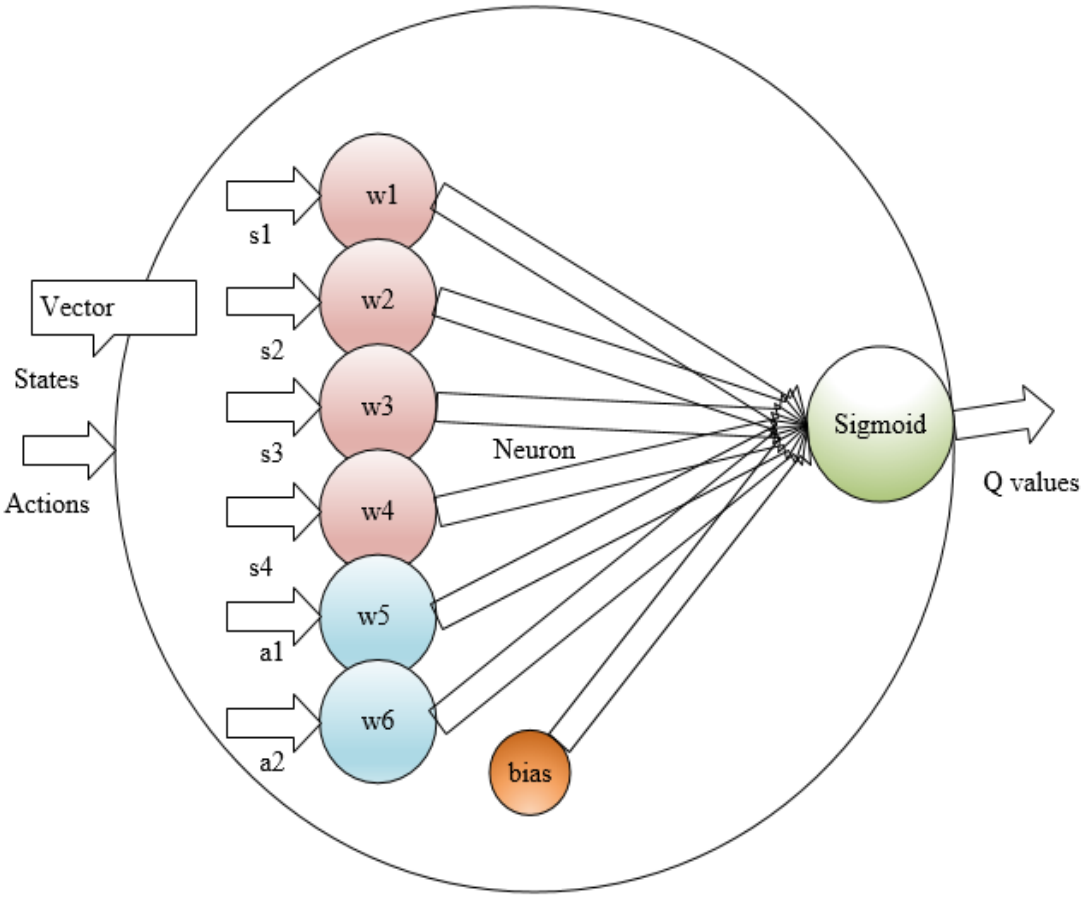

Figure 17. Single Neuron Implementation for Q-learning

Single Perceptron based design is extended to design a Multi-layer perceptron for calculating the target Q values. For this implementation a neural network with 6 input neurons, 4 hidden layers and 1 output layer is implemented. Each of the 6 input neurons are connected to either the state or action. Figure 18 presents the MLP architecture for the implemented algorithm.

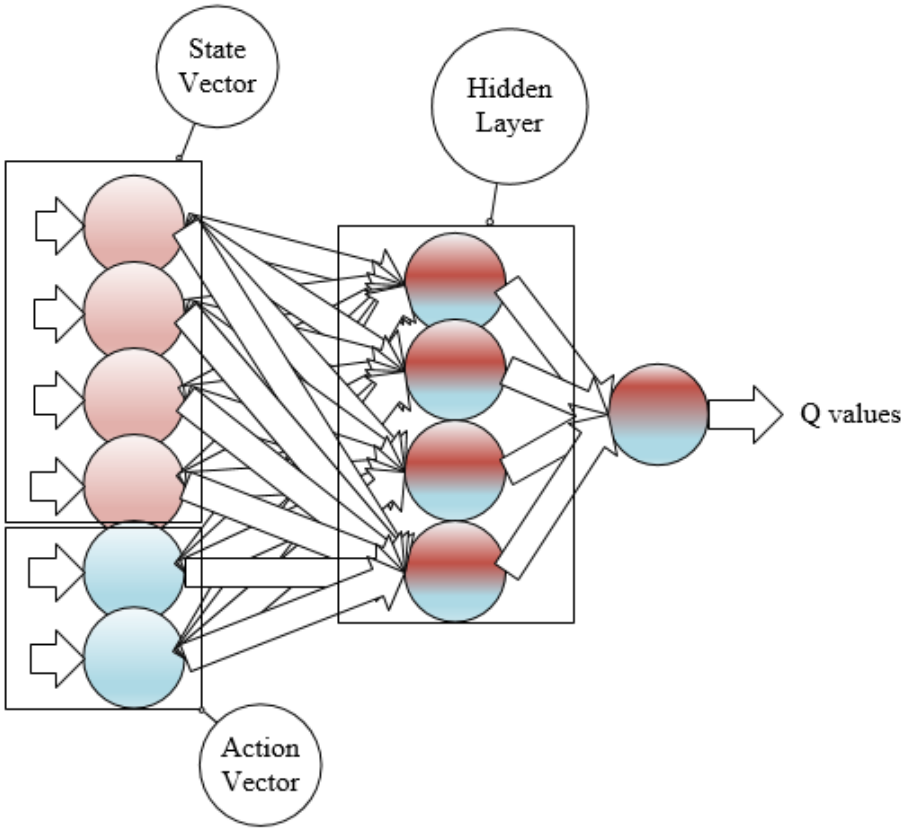

Figure 18. Multi-Layer Perceptron Implementation for Q-learning

The target Q values are calculated based on equations (12-13), with input as vector containing state and action pair. The error is propagated according to equations (14-17). The propagated error is used to update the weight and bias values. The target Q value for propagating error is determined by equation (5). To obtain the target Q value, or the Q error to be back propagated, the feed forward step that constitutes equations (12-13) is run for twice the number of possible actions in a state. Once for calculating Q values for current state and next time to calculate the Q values for future state.

```
Byte Bot                                        // Bot Representation
Double SensorTemp[2]                        // Temperature Sensor Value
Double SensorProx[2]                        //  Proximity Sensor value
Double df                           // Discount Factor
Double lr                           // Learning Rate
STATE0..STATE23 = 1..24



QLL_NN(Q, s, a)
    While(epochs)
        s = GetState()
        While (ActonRuns == ACTIONS)
            Q[runs] = feedForward(s, a[runs])
        endWhile
        Qmax, a = max(Q[runs])
        ActionReturn = PerformAction(a)
        if(ActionReturn != 0) snew = GetState()
        While(ActionRuns == ACTIONS)
            Qfuture[runs] = feedforward(s, a[runs])
        endWhile
        QfutureMax = max(Q[runs])
        r = GetRewards(s, snew)
        QTarget = r + (df*QfutureMax)
        Backpropagate(QTarget)
        s = snew
    endWhile

GetState()
 if(sensorTemp[0] < sensorTemp[1])
        lcount = lcount + 1
        return [STATE0 - STATE7] ?  SensorProx[0], SensorProx[1], Lcount
 elseif(sensorTemp[0] > sensorTemp[1])
        rcount = rcount + 1
        return [STATE8 - STATE15] ?  SensorProx[0], SensorProx[1], Rcount
 elseif(sensorTemp[0] > sensorTemp[1])
        rcount = rcount + 1
        return [STATE8 - STATE15] ?  SensorProx[0], SensorProx[1], Rcount

```

Figure 19. Pseudo Code for Q-learning using Neural Networks.

The pseudo code for Q-learning using neural networks is demonstrated in figure 19. The environment sensing, rewards, action selection and performing an action is same as that of Q table implementation. However, the major difference is the usage of Neural networks for generating and updating the Q values.

3.3. Hardware Accelerator Architecture

Following section demonstrates the architectural implementation of the hardware accelerator for Q-learning algorithm. The flow of demonstration is as follows.

1. Single Neuron Architecture implementation: Starts with a single neuron architectural implementation for supervised learning and using some of the implemented features, move on to implement reinforcement learning (Q-learning Algorithm). Implementation for fixed and floating point is demonstrated with throughput calculations.
2. Multilayer Perceptron Architectural implementation: Starts with the architectural implementation for supervised learning, and proceed with an implementation for reinforcement learning (Q-learning) carrying forward the features of supervised learning. Floating point and Fixed point throughput calculations will be demonstrated.

The above implementations are demonstrated using a node level, layer level and weight level parallel architecture, which does not scale well for a very large deep neural network with more than billion state-space mappings or with a very large number of neurons. Hence a pipelined approach is needed. The architectures are designed from a block level perspective using Synphony Model Compiler toolset, Xilinx System Generator and MATLAB.

3.3.1 Perceptron Q-learning Architecture

As seen in figure 10, a neuron constitutes weights, bias and an activation function. The equation (7) clearly presents a requirement for usage of multiplier and accumulator

(MAC) for calculating the value of net. The ‘net’ value of the neuron is progressed through the activation function to calculate the output or the firing rate of the neuron as in equation (8).

The weights and biases are the parameters of a neuron. Weights are read during the feedforward path to calculate the output of a neuron, and are updated after the backpropagation path. For a fine weight level parallel architecture, a First In First Out memory architecture (FIFO) for each individual inputs has been implemented to store and update the weights in parallel. There are many types of hardware schematic implementations for activation functions [18] [19], out of which Coordinate Rotation Digital Computer (CORDIC) based approach and Look Up Table (LUT) based approach have been considered.

The LUT is implemented using a Read Only Memory (ROM). ROM consists of pre calculated sigmoid values for each of the inputs, and the address for the ROM is generated using the Address Translation block (AT). The size of ROM plays a role in the accuracy of the sigmoid value; an increase in the size of ROM, indicates the increase in the sensitivity range of the input, thereby estimating output with greater accuracy. A more accurate and complex approach can be implemented using a CORDIC block. However, increase in complexity of feedforward path with additional cycles leading to an increase in complexity for synchronization and a reduction of throughput made it advantageous to use ROM over CORDIC. With an increase in number of neurons, ROM size reaches its limit which creates a requirement to use a CORDIC based approach sacrificing the throughput for a greater accuracy. Other approaches like piece wise linear interpolation algorithm [13] can also be implemented to calculate the sigmoid value.

The designed architecture for a single perceptron supervised learning is shown in figure 20, the variables are obtained from equations (7-11). Q-learning in a neural network differs from supervised learning in a fact that the training output values are not correct exemplars, but estimates.

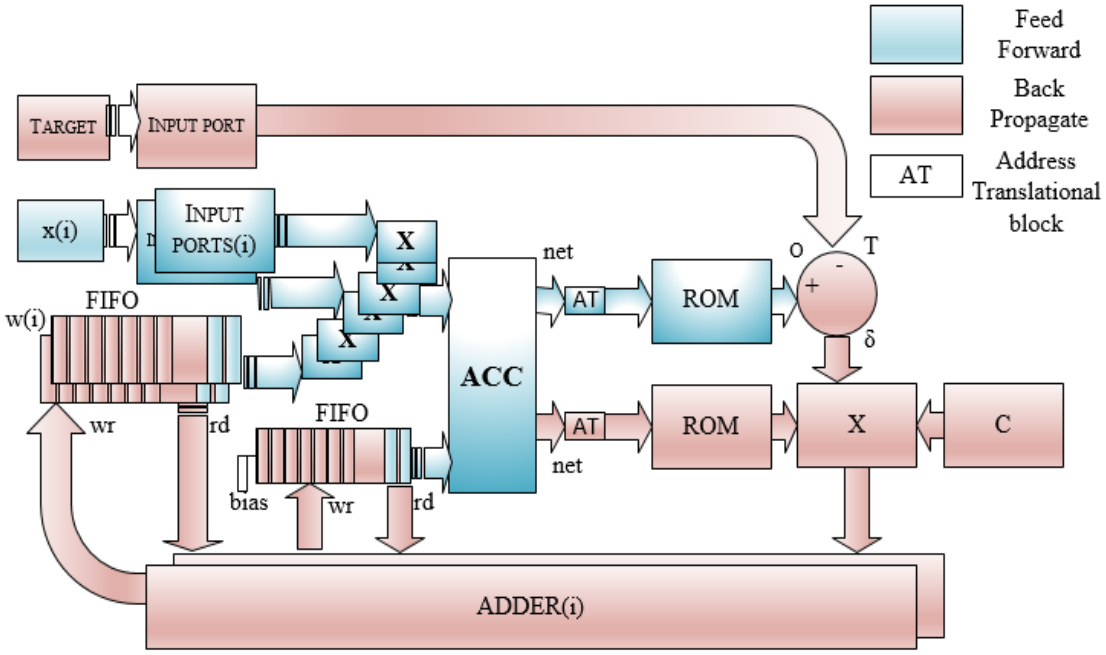

Figure 20. Implemented Perceptron Architecture Schematic for Supervised Learning

The implemented architecture is weight level parallelized. For one pass during feedforward step the output is calculated for all weights and inputs (similar to the state-action space in Q-learning). There are 'w' number of multipliers for each of the input-weight pair. Adder or group of adders (in case of floating point) in feed forward path is used to calculate the value of 'net' as in equation (7). Two ROM's as in figure 20 are used

one for the sigmoid calculation in feed forward path and other for derivative of sigmoid calculation in backpropagation path as in equation (9).

Table 6 gives a peek of the ROM values for sigmoid calculation. An address translation block is used to translate the output net to appropriate ROM address. The output of ROM is an approximated sigmoid of net function approximated accurately by 4 decimal places.

Net	ROM Address	ROM Data
0	0	0.5000
0.0001	1	0.5000
0.0002	2	0.5000
0.00023	2	0.5000
0.5627	5627	0.6371
0.9999	9999	0.7310

Table 6. Peek of ROM with 10,000 Sigmoid Values

Before moving to Q-learning Architecture, an intermittent architecture is presented replacing the fixed target output by a Q table. The architecture in figure 21 demonstrates the use of RAM for storing the Q values. Error is determined by the Q-learning equation (5), with rewards, learning factor and discounting factor provided in the design. Considering the fact that all the Q values are stored in a Q table, the use of such architecture introduces a challenge of using a large size RAM for storing Q table, which was the main reason for using a neural network in first place.

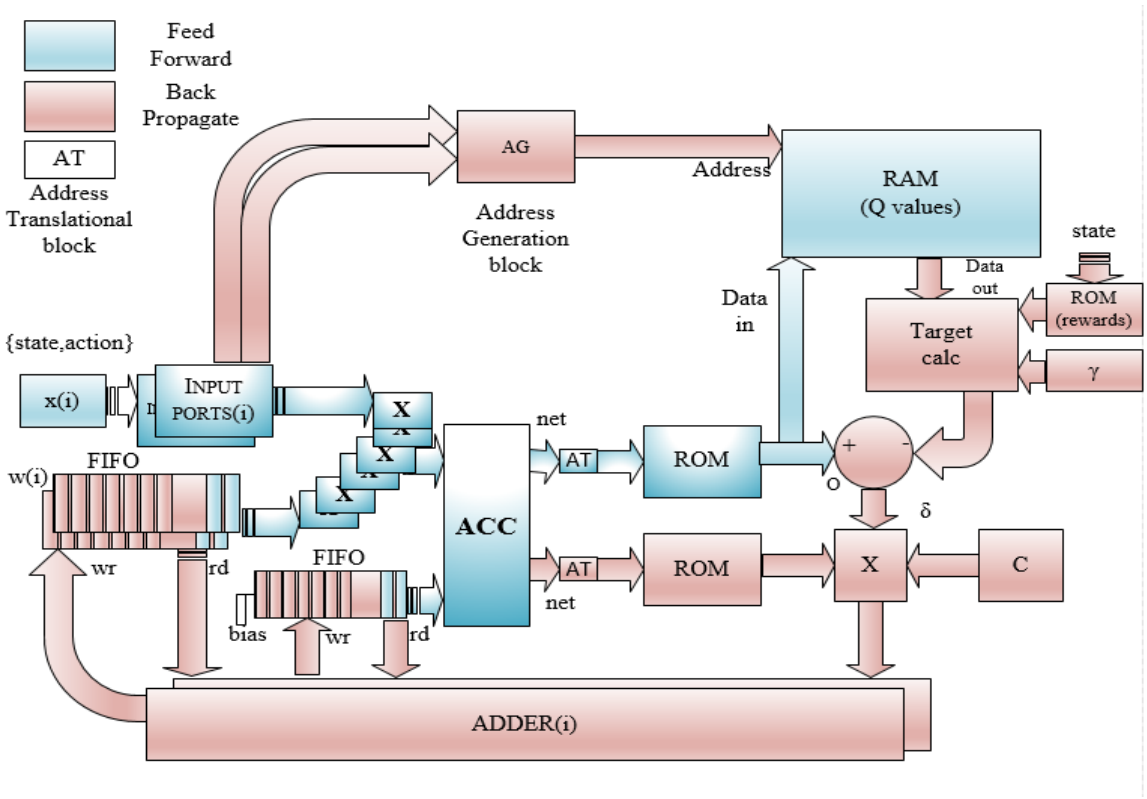

Figure 21. RAM based Architecture for Perceptron Q-learning

In the final architecture presented in figure 22, RAM (that stores entire Q table) is replaced with 2 FIFOs of sizes equal to the number of possible actions in a state. The FIFOs are used to store the Q value for all actions in the current state, and Q values of all actions in the future state respectively. In general, the number of possible actions in a state are very less when compared to the number of total states in a system, hence replacing an action-space sized RAM with an action sized FIFO sounds to be an efficient solution. Calculation of Q values per each action is done using a feed forward pass through the neural network.

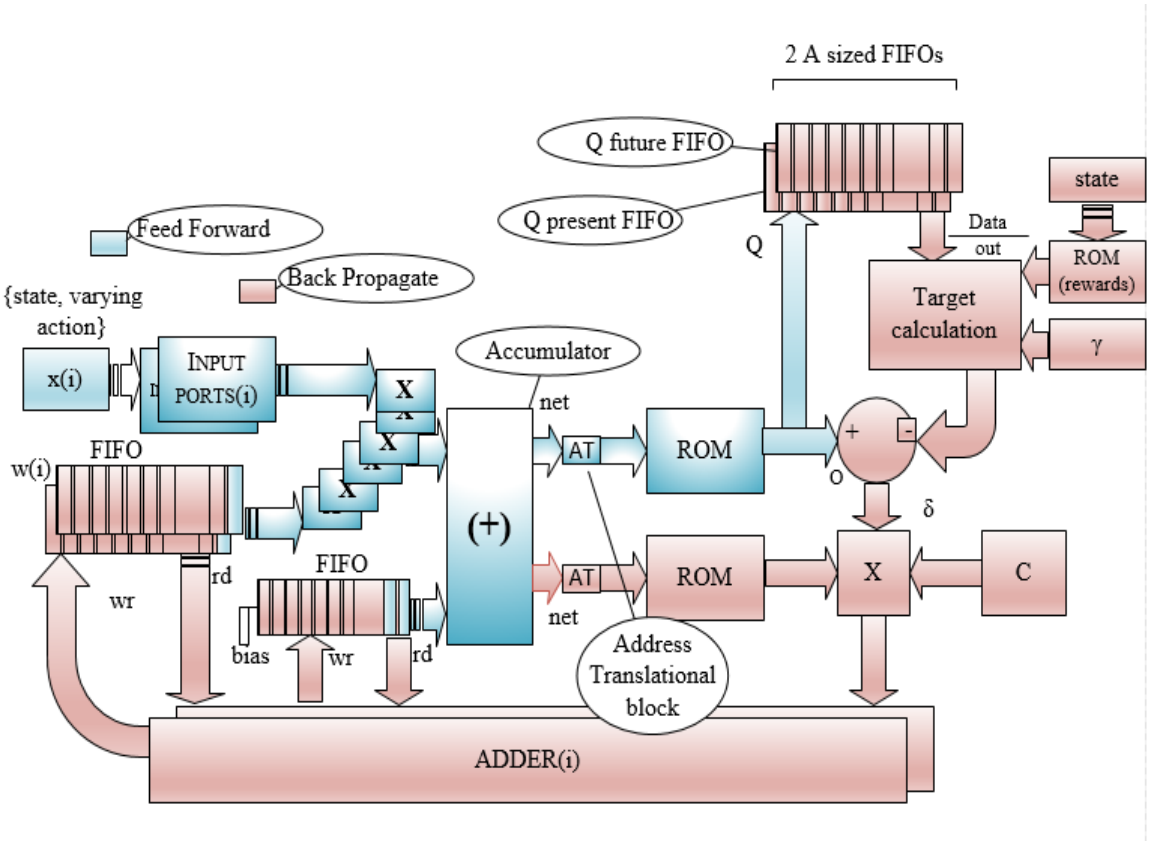

Figure 22. Action Sized FIFO-based Architecture for Perceptron Q-learning

The execution of algorithm is divided into 5 major stages. Transition to one stage from other stage is done through a control bus from a controller. Following are the 5 stages of the algorithm:

1. Calculate the Q values for all actions in a state by executing the feed forward algorithm for A times, where A is the number of possible actions in a state.
2. Select an action a_t, using one of the action selection policies mentioned in section 2.1.1.1, and move to a new state s_{t+1}.
3. Calculate the Q values for all actions in the future state s_{t+1} by executing the feed forward algorithm for A number of times, where A is the number of possible actions in next state.

4. Set the error signal using equation (5) from section 2.1.1.1.
5. Using this error signal, the neural network is trained by propagating error backwards based on equations (9-11).

Control path and data path for the architecture is shown in figure 23. Feed forward stage is enabled for calculating the Q values for each of the 2 stages, i.e. stage 1 and stage 3. Error propagation and backpropagation are enabled after 2 feed forward states for updating the weights and biases.

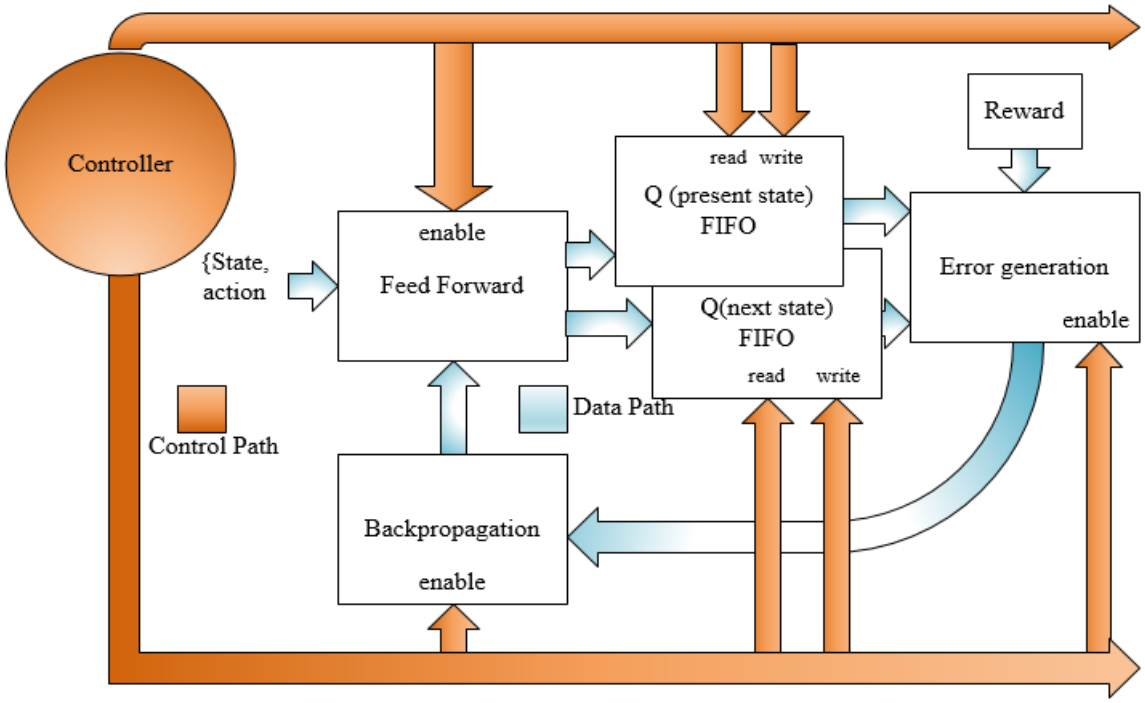

Figure 23. Control and Data Path Representation of the Perceptron Q-learning Architecture

The blocks enabled during stage 1 are shown in figure 24. Two FIFOs are used to store the Q values for current and next state. During stage 1, current state FIFO is enabled

storing the Q values for each of the possible actions. The FIFO for future state is not read or written and is in disabled state, same is the case for blocks in backpropagation.

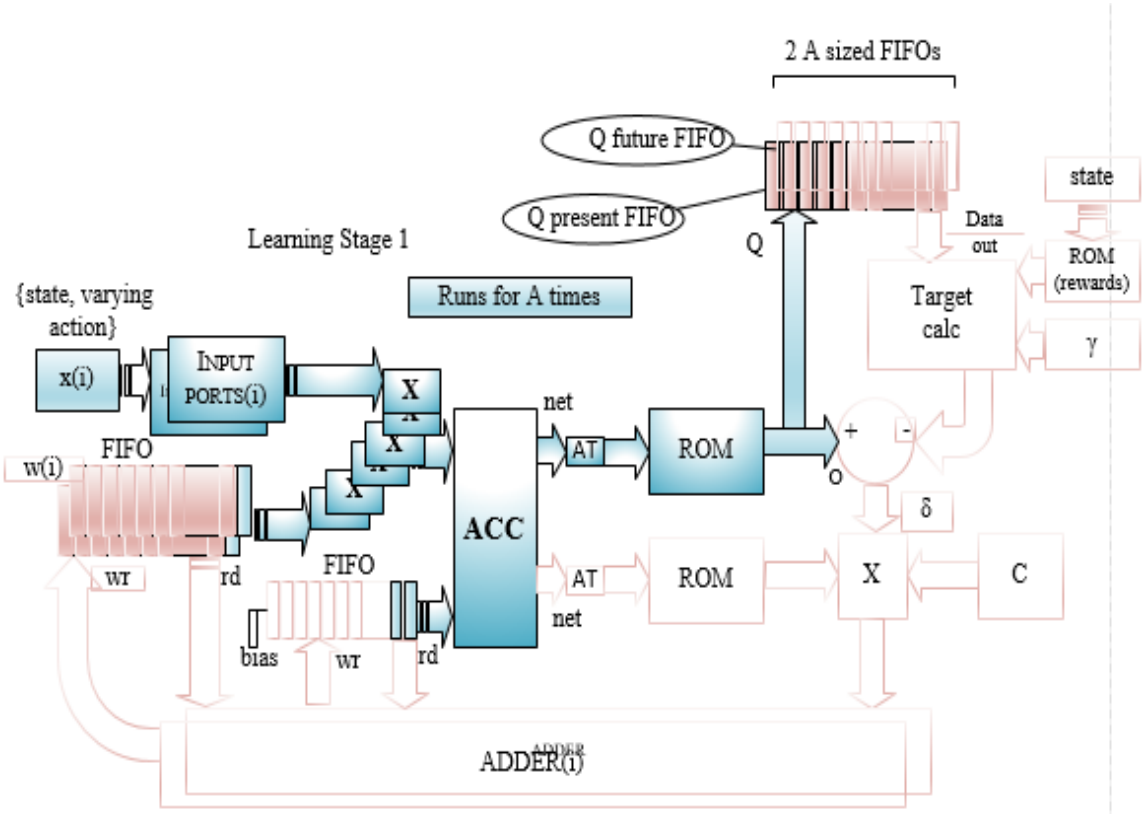

Figure 24. Stage 1 Execution for FIFO based Q-learning Architecture

The stage 3 for the learning architecture as shown in figure 25, is similar to stage 1, the only difference being the usage of different FIFOs based on whether the forward pass calculates Q values of current state or Q values of next state.

The stage 2 is an action selection policy implemented as a MATLAB function, which in real-time scenario would be implemented by a CPU. An action is chosen based on the Q values in the current state. As mentioned in Section 2.2.1, an action policy can be

completely greedy by selecting an action with maximum Q value or 90% greedy, during which an action is selected 90% of its time based on the maximum Q value.

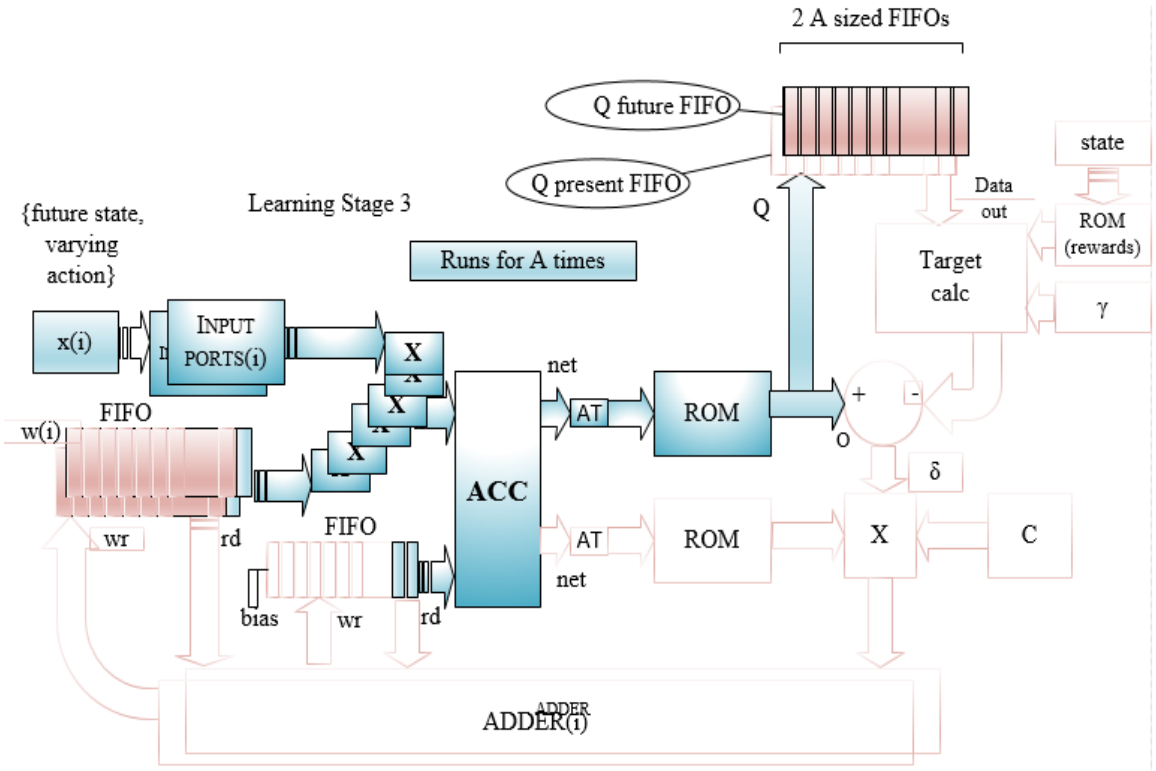

Figure 25. Stage 3 Execution for FIFO based Q-learning Architecture

The error generation is processed during stage 4 and propagated during stage 5 of the algorithm. The error generation step uses equation (5) for calculating error values. Each of the Q values from the future FIFO is read and an optimal (maximum value) is determined, by comparing those values. During stage 5 process, the error calculated using equation (5) is back propagated using equations (9-11). The complete schematic flow for stage 4 and stage 5 is demonstrated in figure 26.

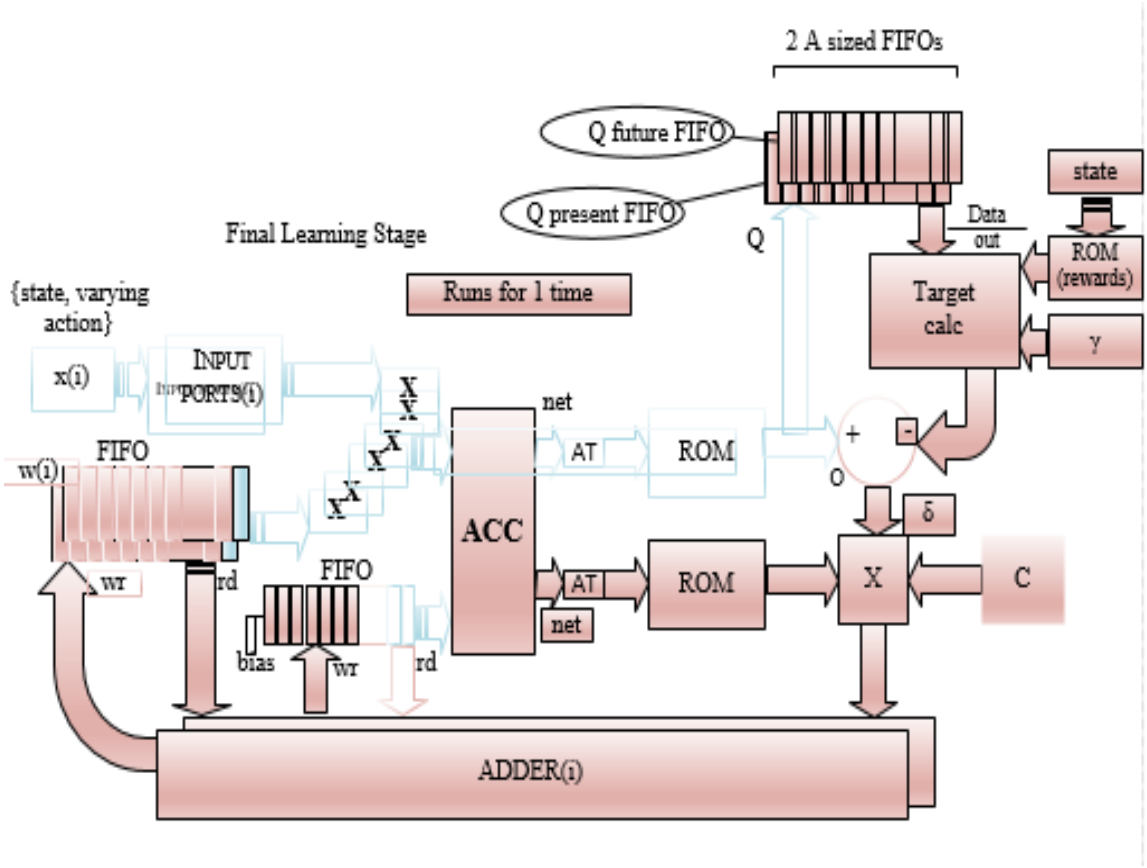

Figure 26. Error Calculation and Propagation Stage for Q-learning

3.3.1.1 Fixed Point Perceptron Q-learning Architecture

Fixed and floating point represent the storage format for numerical data, fixed point implies the presence of fixed number of digits before or after the decimal point. Fixed point parameters in a Synphony Model Compiler (SMC) tool has word length and fraction length as its input parameters. The accuracy and power (resource utilization) tradeoff is done modifying the lengths of word size and fraction size. The number of clock cycles required for each of the fixed point hardware blocks in SMC is presented in table 7.

Hardware	Clocks
FIFO (push,pop), ROM, RAM	1
Input port (read)	1
Multiplier, Adder, Comparator	1
Translation	1

Table 7. Clocks for Fixed Point Blocks

Assuming there are A actions per state and S states in total, according to figure 27, The stage 1 for Q-learning consumes 3 clock cycles for calculating each Q value and storing in FIFO. Therefore, for a single execution of stage 1, the total number of clocks required is equal to 3A.

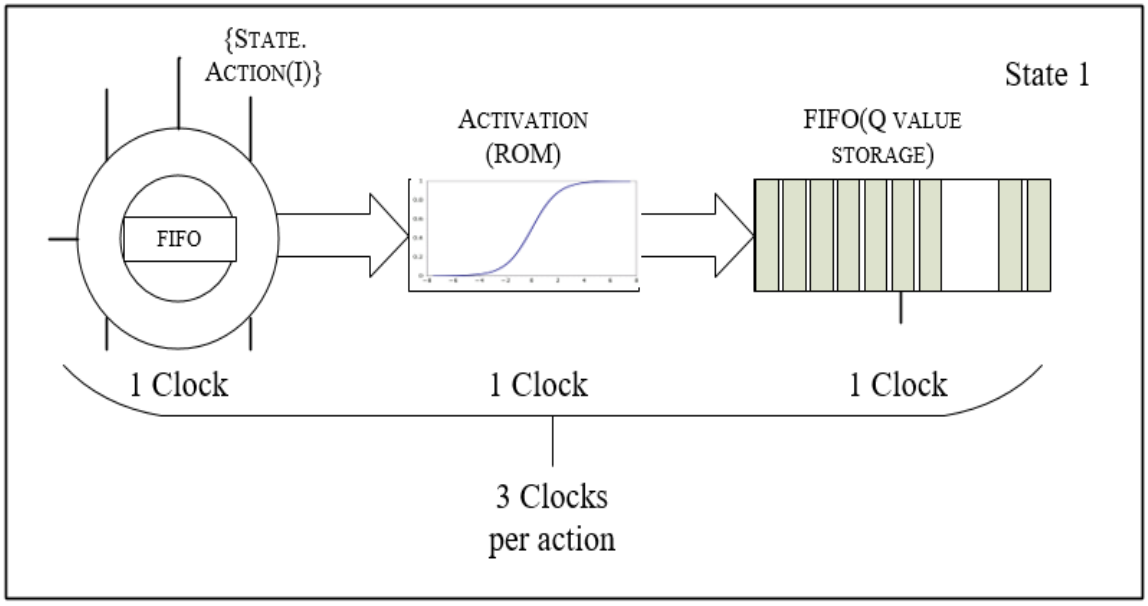

Figure 27. Clocks per action in stage 1 for Fixed Point Perceptron Q-learning

Stage 2 comprises of action selection (state modification) policy. The state modification policy has to be determined real-time and cannot be calculated based on the

existing values. A block with 0 clock cycles delay has been modelled for the action selection policy. Stage 3 and Stage 1 have similar execution strategies. Stage 3 and Stage 1 consume same number of clock cycles and the only variant between two stages is the state-action vector inputs. Therefore, the number of clock cycles for calculating the Q values for the future state is 3A.

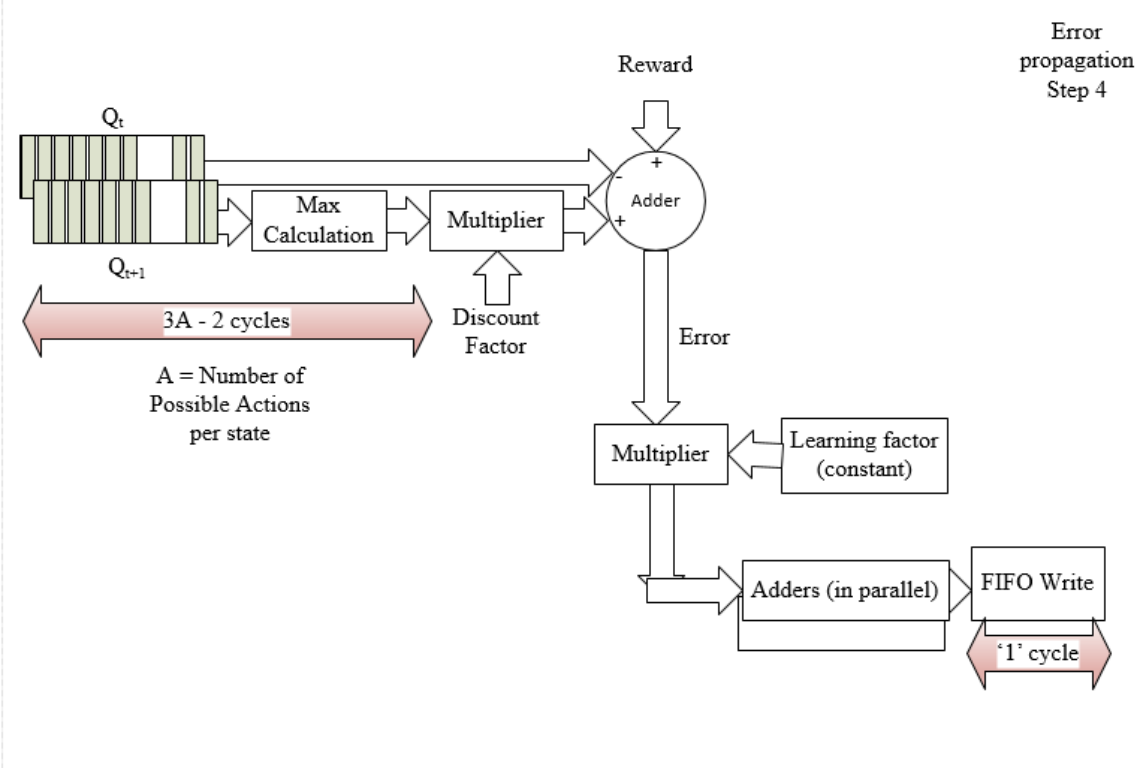

Figure 28. Fixed Point Clocks for Learning Stage in Perceptron Q-learning

The last two stages are combined in a single block diagram shown in figure 28. The error propagation includes calculating the future optimal Q value (based on Qt+1 values in its FIFO), multiplying with the discounting factor and adding the reward value. The error is propagated to update the weights and biases of the neuron, which implies updating the FIFO values. The total number of clocks for updating the FIFO values is the combination

of ‘3A - 2’ cycles for FIFO read, maximum calculation (includes secondary FIFO read/write with comparator) and 1 cycle for updating weights in parallel, totaling to 3A -1 cycles.

Stages	Clocks
Stage 1	3A
Stage 2	0
Stage 3	3A
Stage 4+5	3A-1
Total Cycles/ Q value	9A - 1

Table 8. Fixed Point Clocks per single Q value update in Perceptron Q-learning

The total number of clocks for updating a single Q value is 8A, as in table 8. The throughput calculation as shown in the table 8 is independent of the number of states, however it is dependent on the number of actions per state due to the fact that all possible actions in a state are to be considered for updating a single Q value.

The ROM used in the architecture for calculating the output Q value is preceded by an address translation block and a MAC block, the translation block and MAC block outputs value based on the type of numerical representation of the blocks. For example, figure 29 shows the difference in calculated ROM output for same set of inputs and same ROM data. The reason is tied to the fact that, one of the representation is of floating point and the other is a fixed point implementation. Floating point implementation of the architecture is demonstrated in section 3.3.1.2

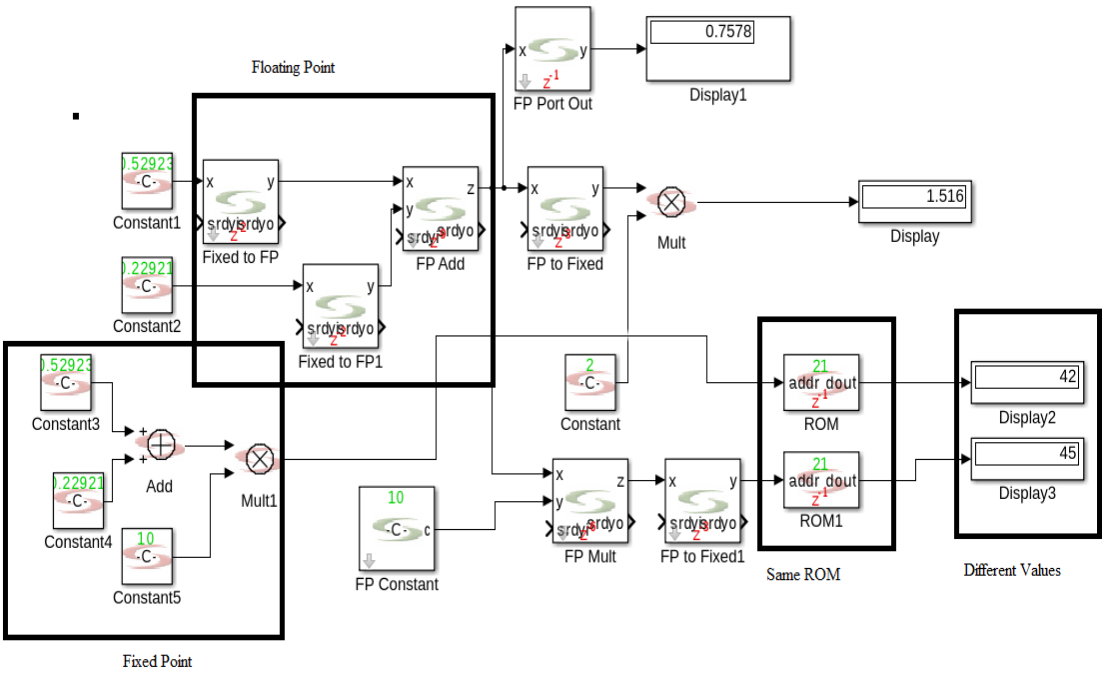

Figure 29. ROM Access Value Differences Between Floating and Fixed Point

3.3.1.2 Floating Point Perceptron Q-learning Architecture

Floating point numbers follow a number representation in the form of $m * 2^e$, where m is the mantissa and e is the exponent. While in fixed point, the gaps between 2 adjacent numbers is always 1, in floating point the gaps between 2 adjacent numbers is not uniformly placed.

The floating point blocks in SMC tool have its clocks dependent on the amount of pipelining involved and the dynamic range of the number. Table 9 shows the number of clock cycles for each of the used blocks

Hardware	Clocks
Input port (read)	3
Multiplier	3
Adder	9
Translation	3
Fixed to Floating	2
Floating to Fixed	3

Table 9. Clocks for Each of the Floating Point Blocks

Throughput calculation for floating point computation differs from fixed point in a fact that the throughput is dependent on the size of the state vector which is not desirable. The reason for the dependence is because of the weight level parallelism implementation of the architecture. However, the state vector and state size are different from each other; the latter (state size) is obtained from the former (state vector size). For example, the implemented rover consists of 24 states, however the size of state vector is equal to 4. Similarly, the number of possible actions is 9, however the size of action vector is equal to 2. Hence, throughput dependence on the size of action and state vector would not affect the performance of implemented architecture.

The number of clock cycles for stage 1 is demonstrated in figure 30. For a single pass, one Q value is calculated and thus for completing the entire stage 1, ‘A’ number of forward passes need to be executed, thus obtaining total number of clocks as ‘A(14+9(s+a))’, where s and a are size of state and action vector

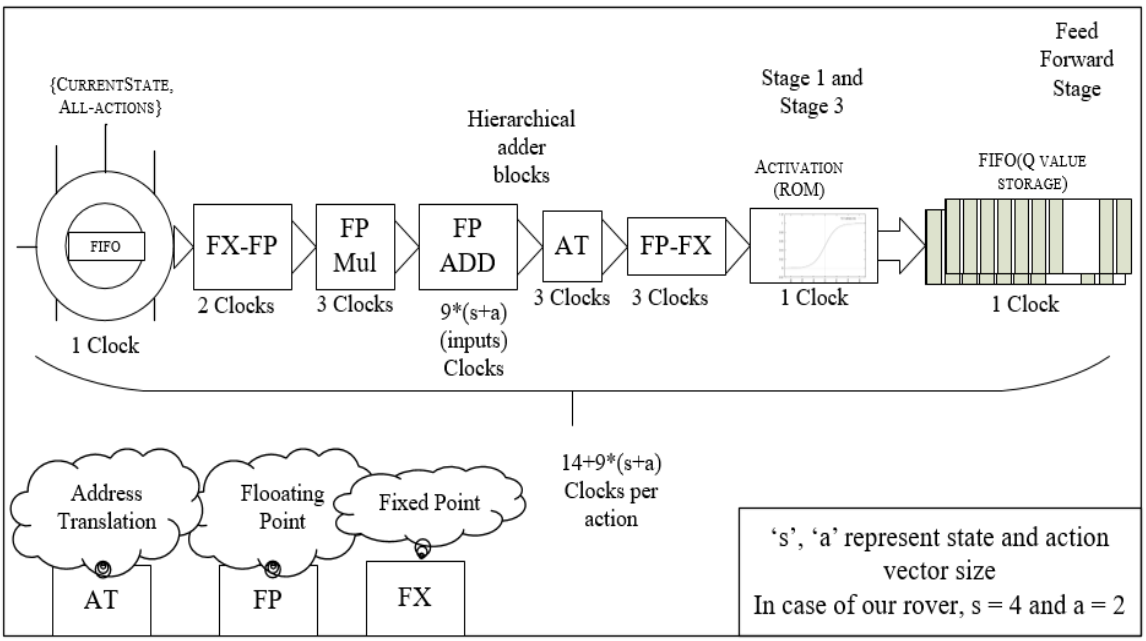

Figure 30. Floating Point Clocks for Stage 1 in Perceptron Q-learning

The step 2 is the action selection policy which is modelled in MALTAB with a zero block delay similar to that of fixed point. An action is selected and performed in stage 2 leading to a stage s_{t+1}. The Q values for the next state is calculated in stage 3, Figure 30 also demonstrates the number of clock cycles utilized for calculating Q values per each action in the future state. The total number of clock cycles in stage 3 is equal to the total number of clock cycles in stage 1.

The final state for learning algorithm is generating error and back propagating it. Each of the Q values in the next FIFO state is compared with each other to find the maximum Q value thereby generating the error according to equation (5). The error is back propagated to update each of the weights in parallel which happens in a single cycle. The total number of cycles required for updating each of the existing weights and biases is equal to '3A + 40' cycles. An observation that can be made here is, that the number of cycles in backpropagation is independent on the number of weights or the input vector size.

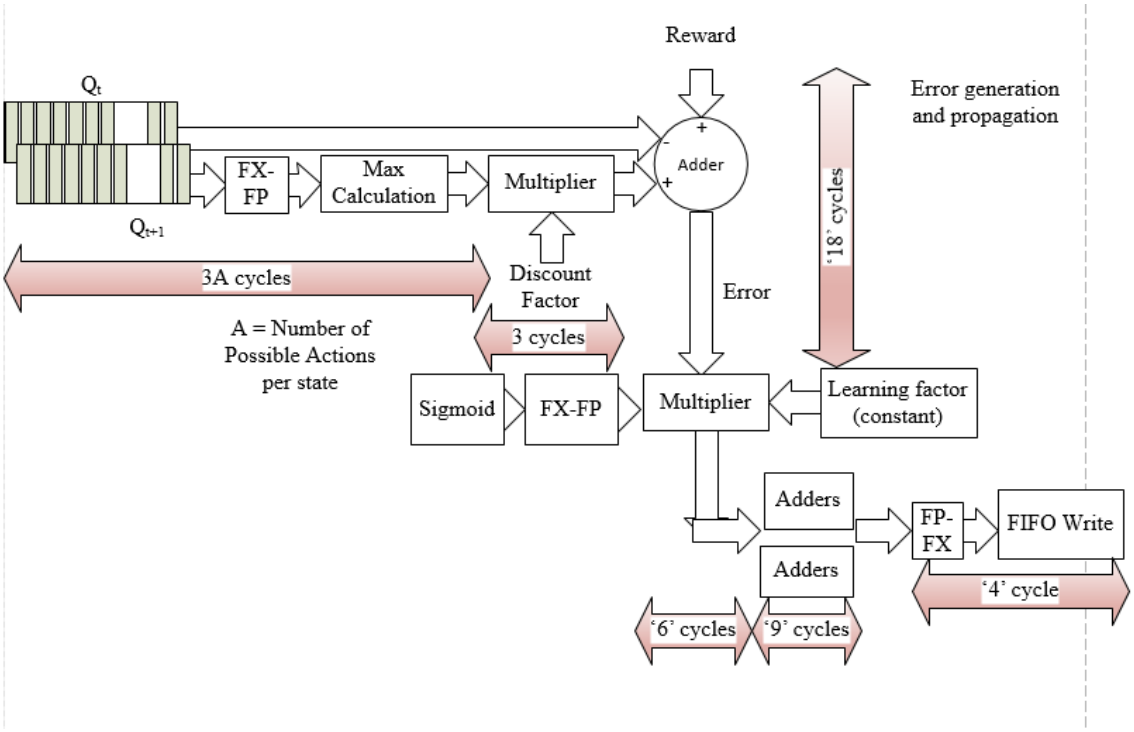

Figure 31. Error Generation and Propagation Stage for Floating Point Perceptron Q-learning

The total number of clock cycles for updating one Q value is obtained from the sum of total clock cycles for stage 1, stage 2, stage 3 and stage 4. The total number of clocks is shown in table 10.

Stages	Clocks
Stage 1	A*(14 +9*(s+a))
Stage 2	0
Stage 3	A*(14 +9*(s+a))
Stage 4+5	3A + 40
Total Cycles/ Q value update	31A + 18A*(s+a) + 39

Table 10. Clocks for Q Value Update in Floating Point

3.3.2 Multi-Layer Perceptron Q-learning Architecture.

Multilayer perceptron is a combination of one or more neurons arranged in layers, with each layer comprising of neurons with same feature as that of a perceptron. The error generated is back propagated to each of the previous layers, updating the weights and biases of the neurons as in equations (12-17).

MLP for Q-learning algorithm consists of an input layer with number of neurons equal to the size of state plus action vector, with one output layer and one or many deep layers. However, for the current accelerator implementation, only one hidden layer is considered. MLP accelerator implementation comprises of each of the neuron designed with the same architecture implementation as that of previous sections. There are few additional blocks named backpropagation included which works quite differently from that of backpropagation algorithm in a single perceptron. Node level, Layer Level and Weight level parallelism is exploited as described in section 2.2.3. This implementation provides the maximum possible throughput an architecture for Q-learning can achieve, due to the exploitation of all types of parallelism. However, such implementation is not suggested for a very large neural network or for a complex environment, for which resource utilization is more of an importance.

The control and data path flow for the MLP Q-learning architecture is demonstrated in high level in figure 31. The data flow process in the above architecture comprises of feed forward step in layer 1, layer 2, layer3, updating the FIFO values for current state and next state, finally error generation and backpropagation through layer 3, layer 2 and layer 1.

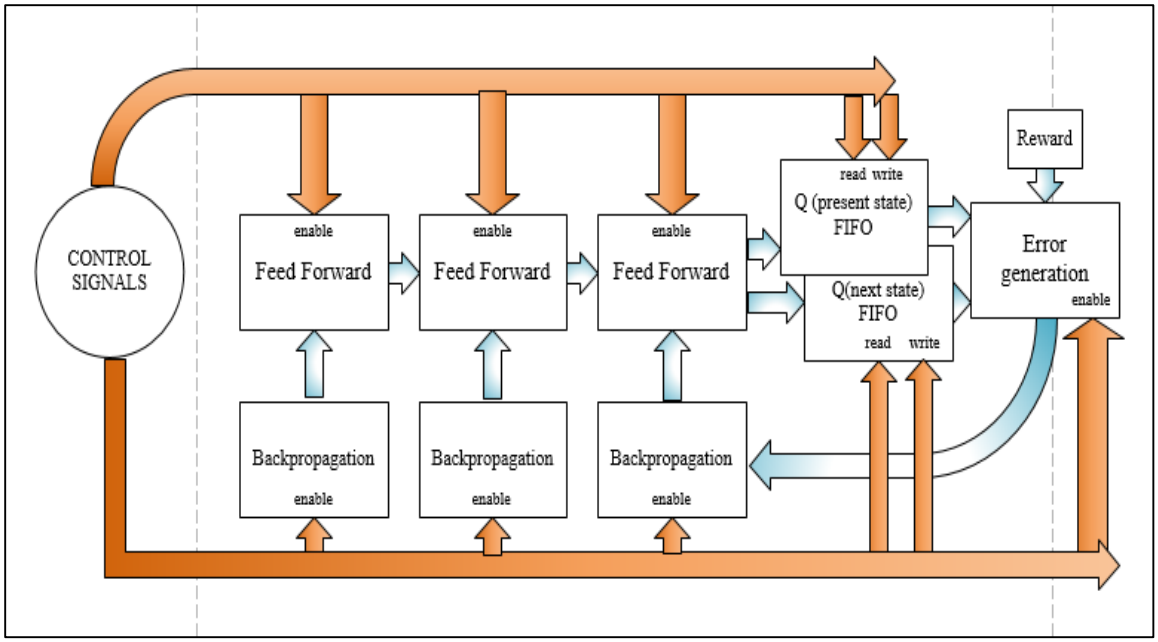

Figure 32. Control and Data path for Q-learning Multi-Layer Perceptron Architecture

The execution of the algorithm is divided into 5 major stages controlled by the control signal generator as shown in figure 32. The 5 stages are as follows.

1. Calculate the Q values for all actions in a state by executing the feed forward algorithm in layer 1 followed by layer 2 followed by layer 3 for 'A' number of times, where 'A' is the possible number of actions per state.
2. Select an action a_t using a 90% greedy policy in MATLAB, and move to a new state s_{t+1}.
3. Calculate Q values for all actions in next state s_{t+1} by executing the feed forward algorithm in layer 1 followed by layer 2 followed by layer3 for 'A' number of times, where 'A' is the possible number of actions per state.
4. Set the error signal as in equation (5) from section 2.1.1.1.

5. Use this error signal to train the neural network by propagating backwards updating weights and biases of each of the input layer as in equations (14-17) from section 2.2.2.

3.3.2.1 Fixed Point MLP Q-learning Architecture

Fixed point implementation of MLP algorithm constitutes same blocks as that of single neuron, however the MLP is not a linearized version of single neuron. The reason is due to the fact that the backpropagation algorithm is different in both the implementations. Three FIFO's from each of the single neuron in layer 1, layer 2 and layer 3 have been eliminated, however the MLP constitutes 3 FIFO's independent of the number of neurons. Stage 1 for the algorithm is demonstrated in figure 33. The number of clocks for a single action is calculated to be 7 clocks in which 2 clocks are for hidden and output layer. This is obtained due to the fact that each of the neurons in hidden and output layer consists of its respective FIFO for storing and updating weights and biases. Stage 1 performs a feed forward computation for 'A' number of times, where A is the number of possible actions in a state. This results in 7A clocks for completing the computations in stage 1.

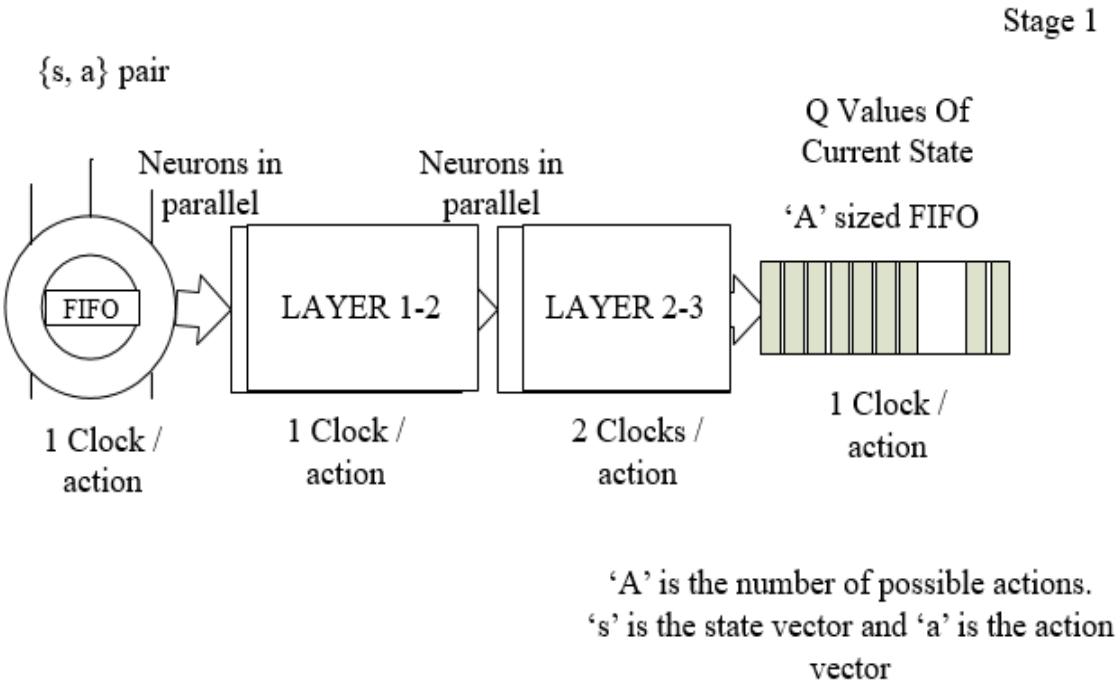

Figure 33. Clocks per Action in stage 1 for Fixed Point Q-learning Multi-Layer Perceptron

Stage 3 is similar to Stage 1, only differing in the fact that a new state also called as the future state is selected as the state vector and Q values of all possible actions in the future state is computed and stored in a secondary FIFO. Figure 34 demonstrates the clocks for stage 3. It takes 7 clocks per action to compute Q values, and 7A clocks to complete the execution of stage 3

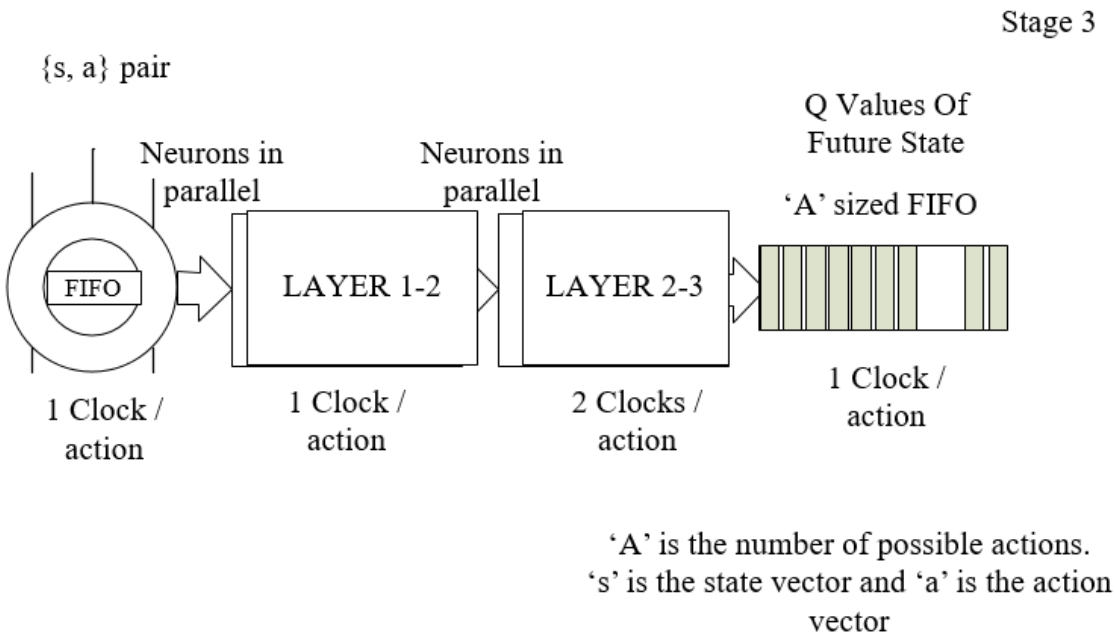

Figure 34. Clocks per Action in stage 3 for Fixed Point Q-learning Multi-Layer Perceptron

Backpropagation and error generation is more complicated in a MLP when compared with backpropagation of the perceptron. Backpropagation comprises of blocks for δ generation and ΔW generation as in equations (14-17). In case of fixed point, for calculating the throughput of fine grained parallel architecture, only the blocks with at least 1 cycle delay like the ROM, FIFO, etc. are considered while the blocks with no cycle delay, like the multipliers and adders are ignored.

Backpropagation begins after calculating Q values for current state and next state for all possible actions. As demonstrated in figure 35, the Q values from next state and Q values from current state, along with the discounting factor and reward function, is used to calculate the error value.

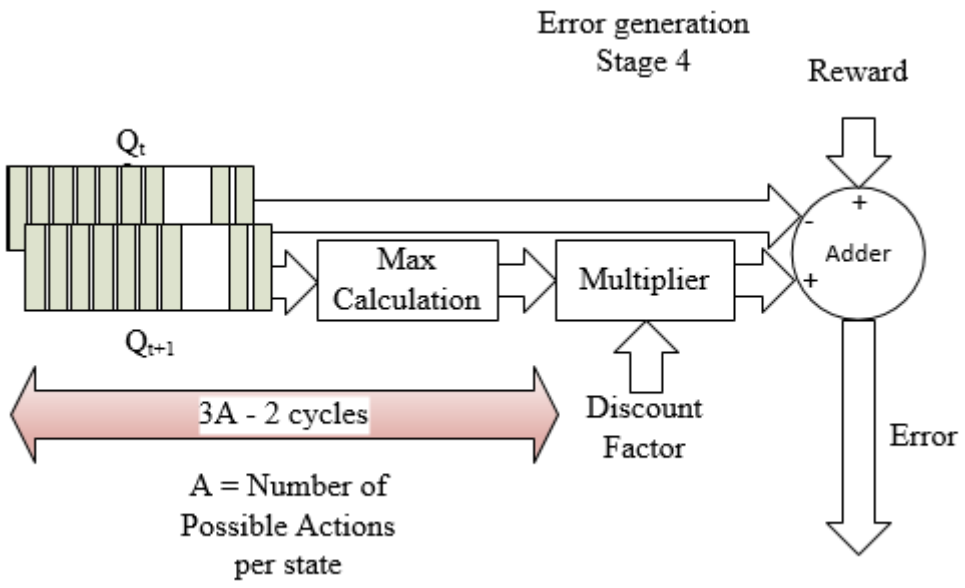

Figure 35. Fixed Point Error generation in Q-learning Multi-Layer Perceptron

The backpropagation in the MLP is sequential in nature. Errors for each of the neuron in a layer is computed, then the error computation is done in parallel by duplicating multiple resources, however the propagation of error is done sequentially, layer by layer. The output error for each of the neuron in a layer is calculated in parallel as shown in figure 36. The weights of each of the neuron is updated using the error values (using existing weights) propagated backwards as in equations (16-17).

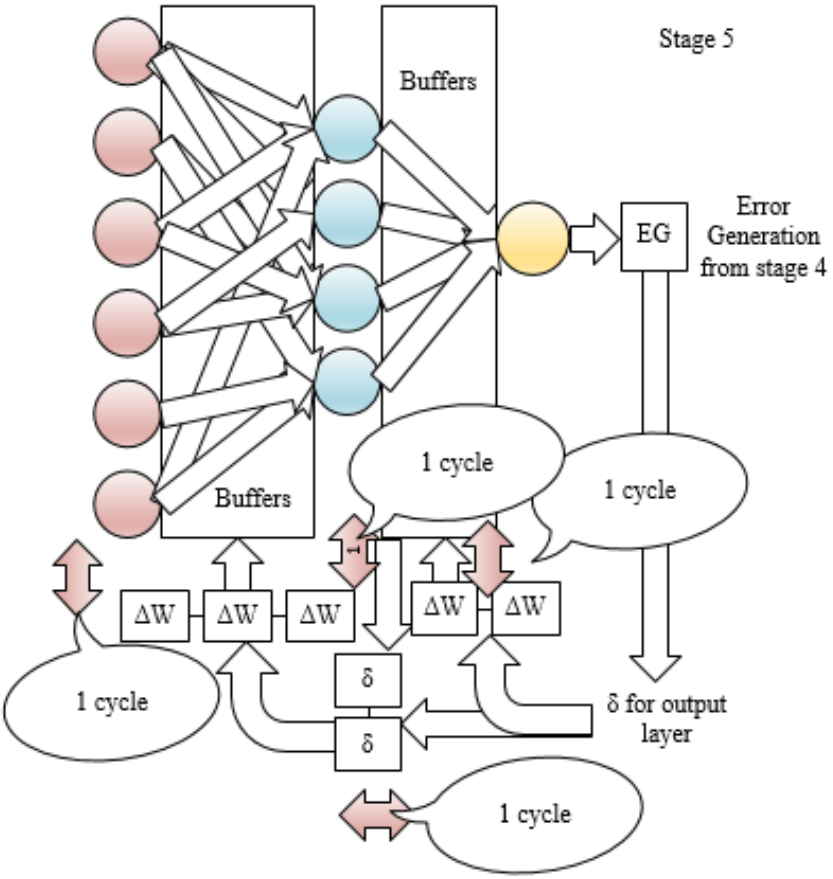

Figure 36. Fixed Point Backpropagation for Q-learning Multi-Layer Perceptron

The δ computation block for hidden and input layers consists of MAC and a ROM consisting of derivative of sigmoid values. There number of blocks for computing δ is equal to the number of neuron-neuron mappings, hence all the δ values are computed parallel using equation (15) for each layer. The total number of clocks for computing δ is 1, and the total clocks for reading and writing weights is also equal to 1. The total number of clocks to complete the stage 5 is calculated to be 4 clocks.

The total number of clock cycles to update the weights of all neurons in the MLP is equal to sum of clock cycles in stage 1, clock cycles in stage 3, stage 4 and stage 5. The total number of clocks is presented in the table 11.

Stages	Clocks
Stage 1	5A
Stage 2	0
Stage 3	5A
Stage 4	3A – 2
Stage 5	4
Total Cycles/ Q value	13A + 2

Table 11. Clocks for Updating Single Q value in a Fixed Point Q-learning Multi-Layer Perceptron

3.3.2.2 Floating point MLP Q-learning Architecture

Floating point design has slightly more complexity involved when compared to fixed point design. The throughput calculation for a fine grained parallel implementation of the algorithm constitutes the presence of H factor, where H is the number of hidden layer neurons. This is undesirable, however, layer level pipelining implementation of such architecture eliminates the presence of 'H' in throughput calculation.

Figure 37, demonstrates the clock cycles for a feed forward computation in stage 1. Q values for each of the actions in a state space is computed by running the feedforward algorithm for 'A' times changing the action vector for each run. The total number of clocks per action for stage 1 is computed to be 28+9*(s+a+H), and thus the total number of clocks per completing stage 1 is A*(28+9*(s+a+H). The total number of clocks for stage 3 is demonstrated in figure 38. The only difference we observe from stage 1 and stage 3 is the usage of different A sized FIFO's.

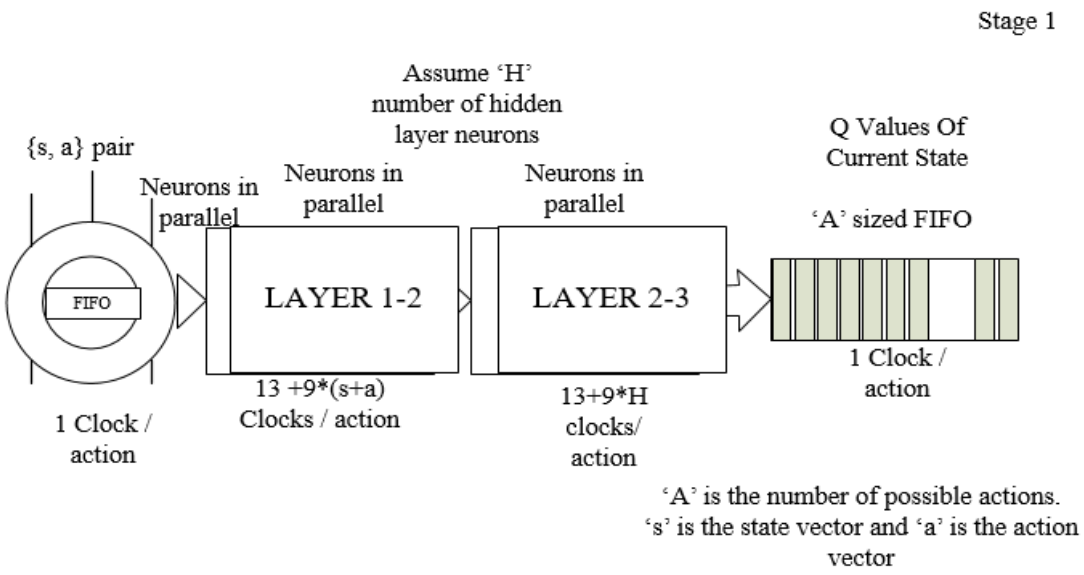

Figure 37. Clocks per Action for Stage 1 in a Floating Point Q-learning Multi-Layer Perceptron

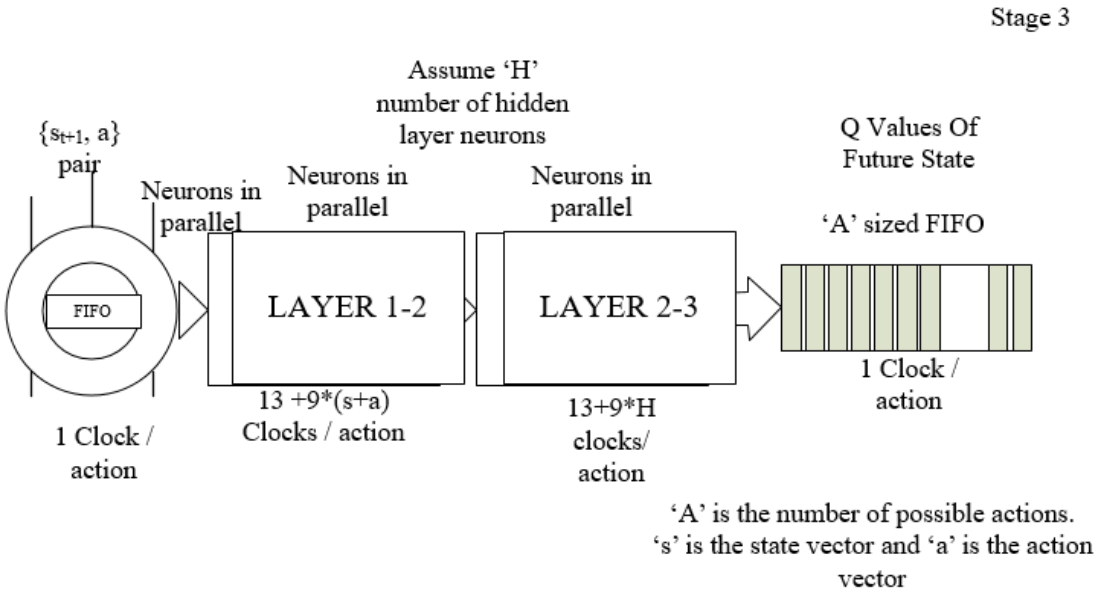

Figure 38. Clocks per Action for stage 3 in a Floating Point Q-learning Multi-Layer Perceptron

The error generation and propagation is more complex in a floating point architecture. The error generation constitutes the maximum Q value calculation in next state, reward calculation and using equations (14-15) to calculate the error value.

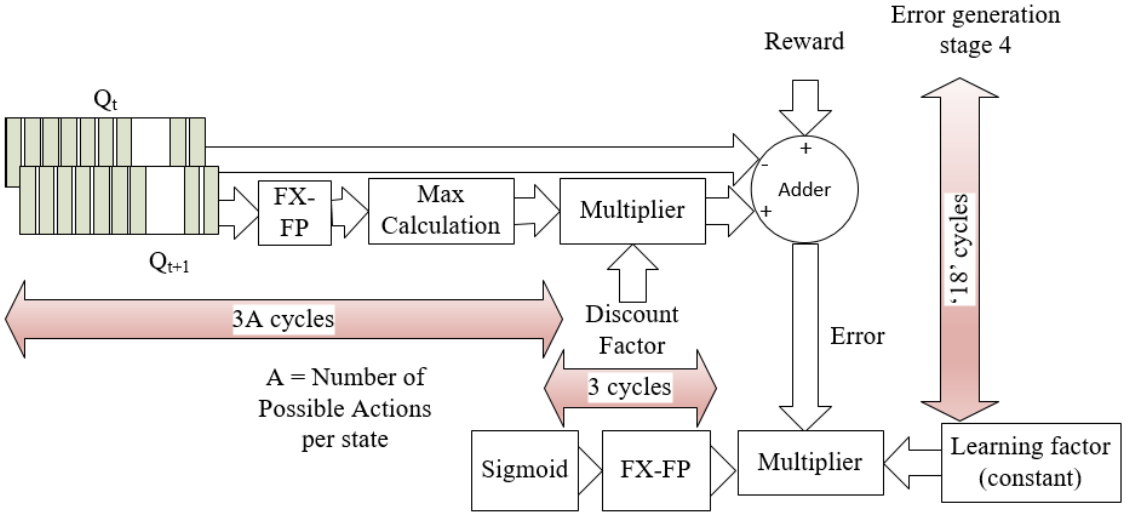

Figure 39. Error generation step in Stage 4 of Floating Point Q-learning Multi-Layer Perceptron

The backpropagation algorithm is implemented using multiple blocks generating δ values and ΔW values of equations (16-17). Figure 40 demonstrates the block level outline for the error propagation step. The block δ generator, uses the equation (13-14) and Δ generator uses the equation (15) for updating the weight values for each of the output, hidden and input layer neurons.

Table12, demonstrates the total cycles in updating the Q value for the implemented floating point architecture. Note that the cycles are dependent on the size of state vector, size of action vector, number of actions, and the number of hidden layer neurons.

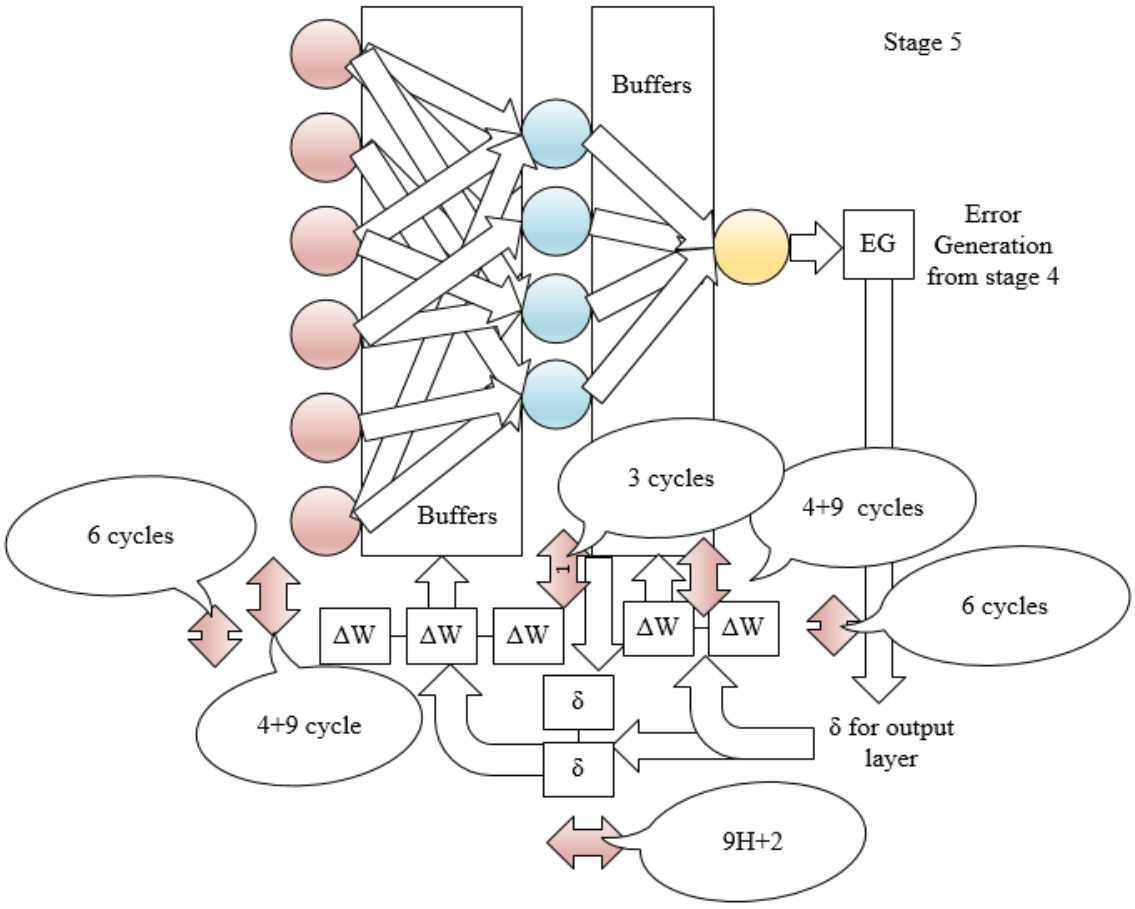

Figure 40. Error Generation and Propagation Representation for Q-learning MLP Floating Point

Stages	Clocks
Stage 1	28A*(9+s+a+H)
Stage 3	28A*(9+s+a+H)
Stage 4	21 + 3A
Stage 5	9H + 43 + sync
Total Cycles/ Q value	507A+64+56A(s+a+H)+9H+sync

Table 12. Total Cycles per Single Q Value update using Floating Point Q-learning MLP

Fine grained parallel architecture with node level, layer level and weight level parallelism has been discussed in this chapter. The architectures have a very high throughput due to the utilization of multiple resources in parallel. A need for a pipelined implementation and plan of implementation is discussed in conclusion and future work chapter.

This section is followed by the Results and Discussions, demonstrating the mobile robot simulator simulation results, FPGA Synphony model compiler simulations and power consumption for various Q learning architectures discussed.

CHAPTER 4

RESULTS AND DISCUSSION

This chapter presents the CPU simulation results of the proposed rover, followed by the results of hardware accelerator implementations. The CPU simulations of the rover movement are performed using Mobile Robot Simulator Tool and visual basic script, with multiple start points of a rover in an environment. The temperature sensors are simulated using a target evaluation process, with position of the rover and position of the target mark as its input parameters.

FPGA simulations are performed using Xilinx System Generator and Synphony Model compiler and utilization results are collected using Synplify pro. The Q value generation and update are simulated in the same CPU, on which the Mobile Robot Simulator simulations are performed.

4.1 Mobile Robot Simulator Simulation.

Q-learning using Q table, single perceptron and Multilayer perceptron is simulated using Mobile Robot Simulator. Number of collisions before reaching a target has been chosen as a performance metric for comparing each of the algorithms. Table 13 presents the constant values considered for the learning algorithm. Figure 41 shows the results for each of the Q-learning implementations.

Discounting factor	0.6
Learning factor	0.8
Neural network learning factor	0.9

Table 13. Q-learning Algorithm Constants

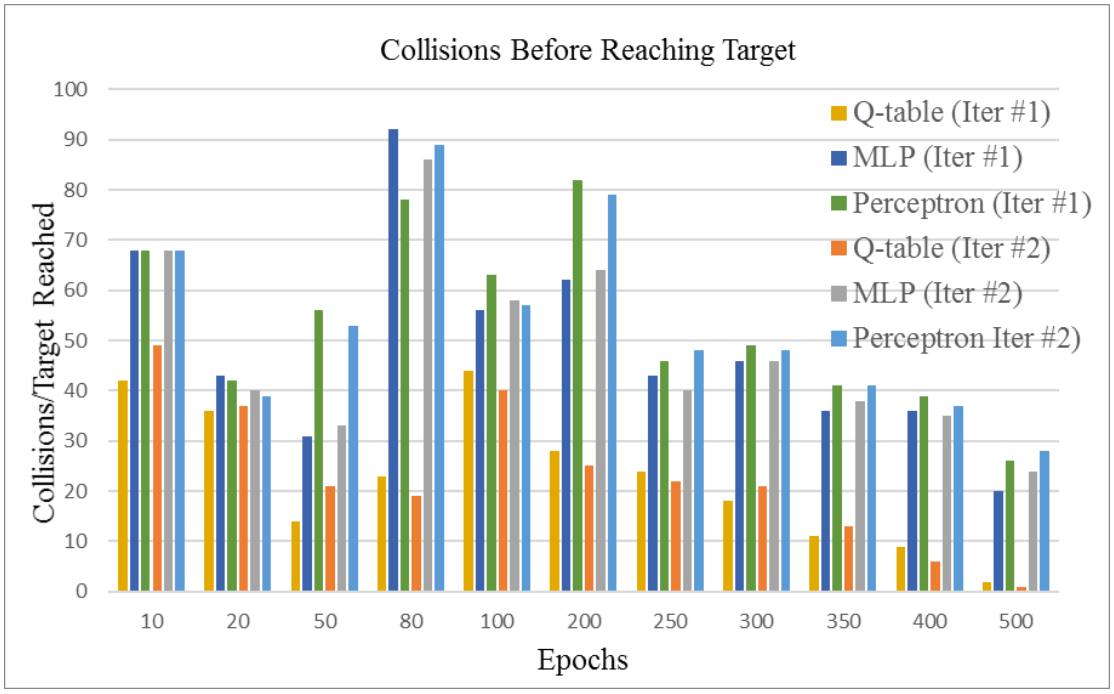

Figure 41. MobotSim Simulation Results for Various Q-learning Algorithms

We observe that the simulation using Q table is performing well compared to simulations using neural network, the reasons can be attributed to the fact that the rover has been simulated only in a simple environment, and with a very low number of possible actions. Q table is of size 216 Q values, for the simulated environment. However, this would not be the case for a rover in realistic complex environment with a very large number of Q values. Moreover, as mentioned, size of Q table grows exponentially with increase in state and action vector size, which acts as a bottleneck for complex simulations. Figure 42, shows the path traversals for each of the Q learning algorithms. The path tracings have been enabled from 400-450 Epochs.

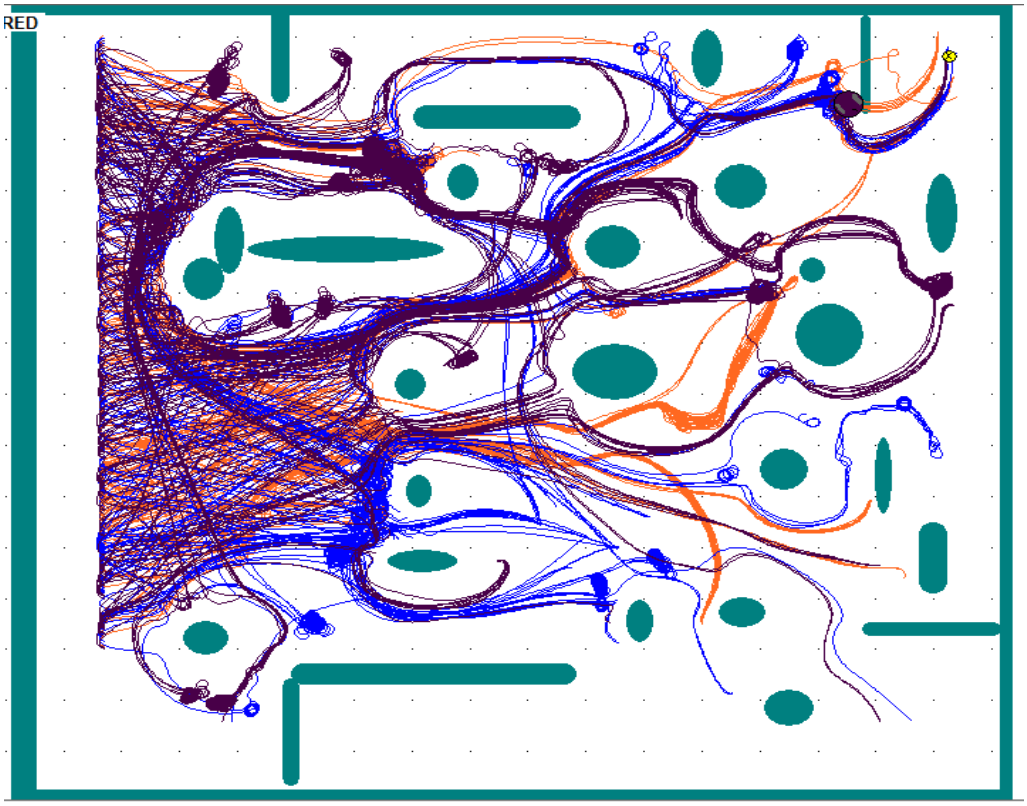

Figure 42. Bot traversals for various Q-learning algorithms

4.2 Synphony Model Compiler Simulations

Each of the presented architectures are simulated using Xilinx Tools on Vertex 7 FPGA. The following combinations of architecture and environment that have been considered.

1. Single Neuron in a simple environment
2. MLP in a simple environment
3. Single Neuron in a complex Environment
4. MLP in a complex environment

Simple and complex environment varies by the fact that the simple environment has a small size of state, action vector and number of possible actions per state. In our case

the size is equal to 6 with size of state vector equal to 4 and size of action vector equal to 2. The complex environment is modelled with combined size of state and action vector equal to 20, possible number of actions per state as 40, and the state space size as 1800. The neural network architecture for MLP consists of 11 neurons in simple environment and 25 neurons in complex environment with 4 hidden layer neurons.

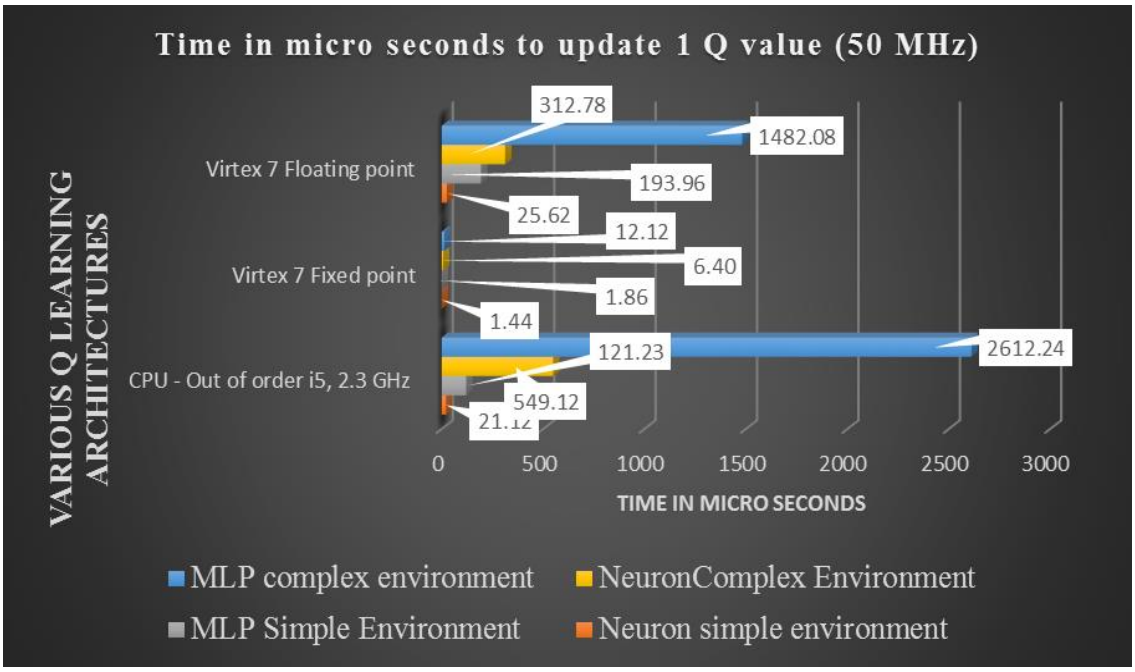

Figure 43. Performance Values for Varying Environments in Q-learning Architectures

Figure 43 presents the performance results for each of the architecture implementations. The Fixed point parallel architecture seems to have a very high performance related to the time to update each of the possible Q value, However, the Mean Square error shows otherwise. The fixed point word length and fraction length plays a major role in trading off accuracy with the utilization (power consumption). Based on this fact, a fixed point architecture can be implemented with a high accuracy, same throughput or performance metric as that of figure 43, while having an increased power consumption.

Power consumption is calculated based on the utilization percentage (one of which is shown in figure 44) of each of the individual resources using Xilinx power estimator tool (figure 45 and figure 46).

Area Summary			
I/O ports (io_port)	258	Non I/O Register bits (non_io_reg)	8049 (1%)
I/O Register bits (total_io_reg)	0	Block Rams (v_ram)	15 (1030)
DSP48s (dsp_used)	17 (2800)	LUTs (total_luts)	12253 (4%)
Detailed report		Hierarchical Area report	

Timing Summary			
Clock Name (clock_name)	**Req Freq (req_freq)**	**Est Freq (est_freq)**	**Slack (slack)**
clk	50.0 MHz	70.1 MHz	5.743
Detailed report		Timing Report View	

Optimizations Summary			
Retiming	0 / 0 more	Combined Clock Conversion	1 / 0 more

Figure 44. Utilization values for Perceptron Q learning in Simple Environment

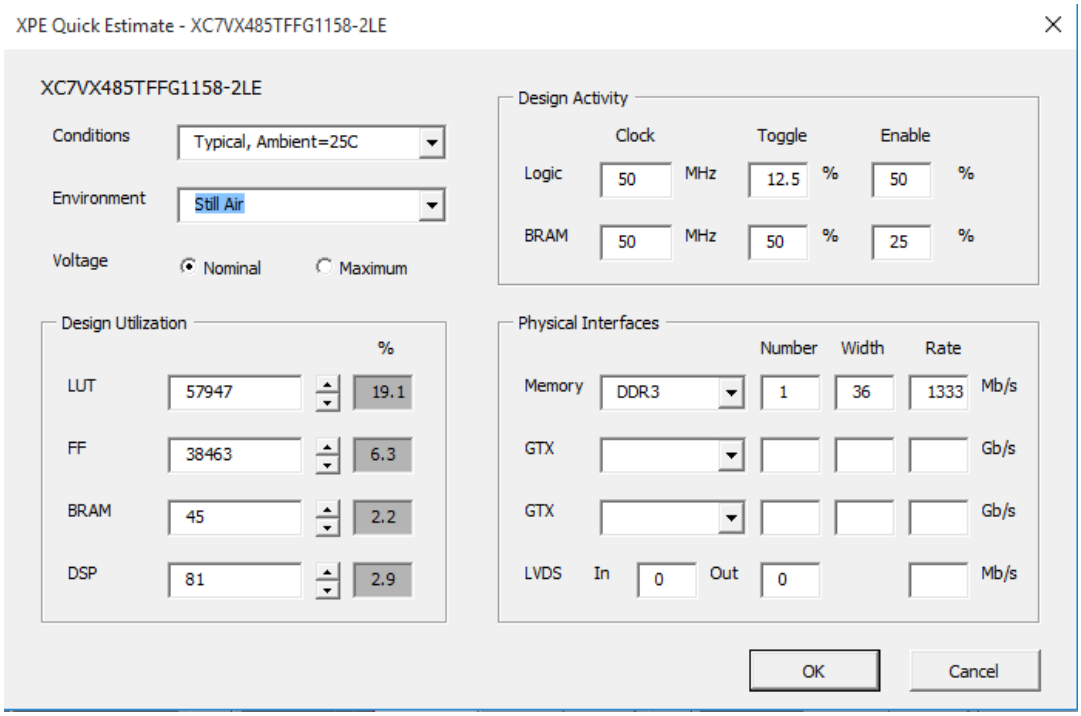

Figure 45. Quick Estimate tool using utilization values

The presence of FIFO to buffer the storage of weights, biases and Q values enables the usage of Block Ram (BRAM). The total on chip power values for each of the implemented architecture is presented in the figure 47, the total on chip power is the

combination of core dynamic power, Input Output (I/O) power, transceiver power and static power of the device. We observe in figure 47, that the peak power consumption is high for floating point architecture running in a complex environment at 50 MHz frequency. Though power estimation is an important factor for consideration, however the energy values is what that is most useful for comparisons, since Q values change over time with improving accuracy, the total time taken to calculate the optimal Q values vary for each of the architectures. The total time taken for finding the optimal Q values can only be obtained, when the learning algorithm is implemented in an actual rover and is exposed to simple and complex environment.

Logic Clock	50 MHz
BRAM Clock	50 MHz
Memory	36 width DDR3 at 1333 Mb/s
Temperature	25C
Environment	Still Air

Table 14. Parameters used in Xilinx Power Estimator Tool

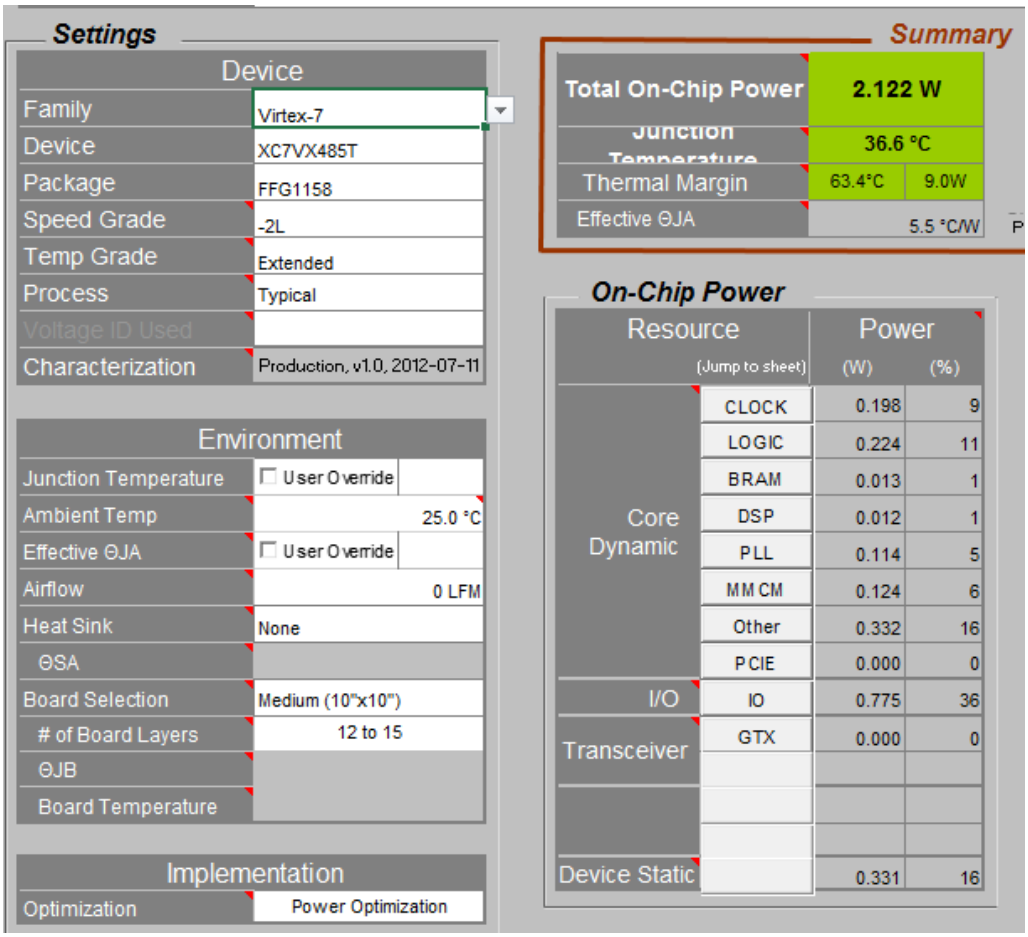

Figure 46. Power estimator tool estimating power value for Floating point MLP Q learning

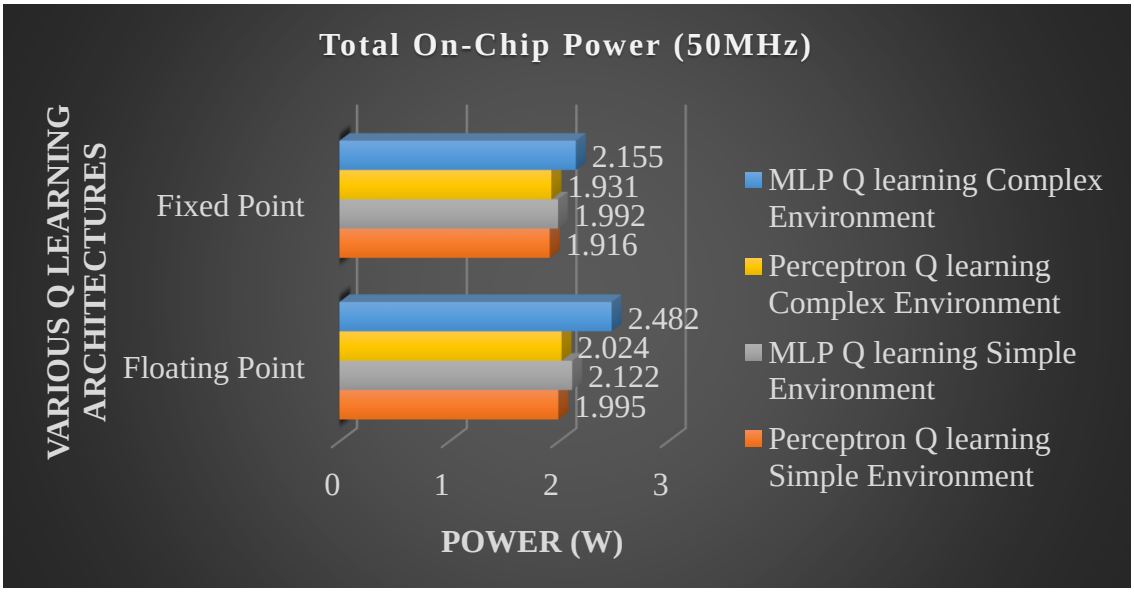

Figure 47. Total on chip Power Consumption for Various Architectures

CHAPTER 5

CONCLUSION AND FUTURE WORK

Reinforcement algorithms, like Q-learning are emerging due to the benefits they provide when combined with deep neural networks. These developments pose a need for accelerating such algorithms and considering the usage of such implementations in robotic applications, the accelerators need be energy efficient. Node Level and parallel hardware architecture for Q-learning using Multilayer perceptron and single perceptron has been demonstrated. An improving performance benefits have been observed with an increase in the complexity of the environment when compared against an out of order CPU. The total on chip power consumption for each of the implemented architectures are calculated using the Xilinx Power estimator tool using the synthesized resource utilization values obtained for Virtex 7 FPGA. The high power consumption is due to the fact that node level, weight level and layer level parallelism has been exploited in combination.

As a part of my future work, I would work on the following aspects

1. Implementing pipelining for each of the architectures.
2. Implementing a piece wise linear interpolation function [13] instead of ROM for calculating the activation function,
3. Introducing the concept of action-replay mechanism in deep neural networks [6].

Pipeline implementation, reduces the resource utilization when compared to the fine grained parallel implantation of the architectures. A combination of pipelining and parallelism like, node level parallelism with weight level pipelining, layer level parallelism with weight level pipelining, and node level pipelining with layer level parallelism are

some of the potential implementations, which can be implemented. Implementation strategy for weight level pipelining is shown in figure 48.

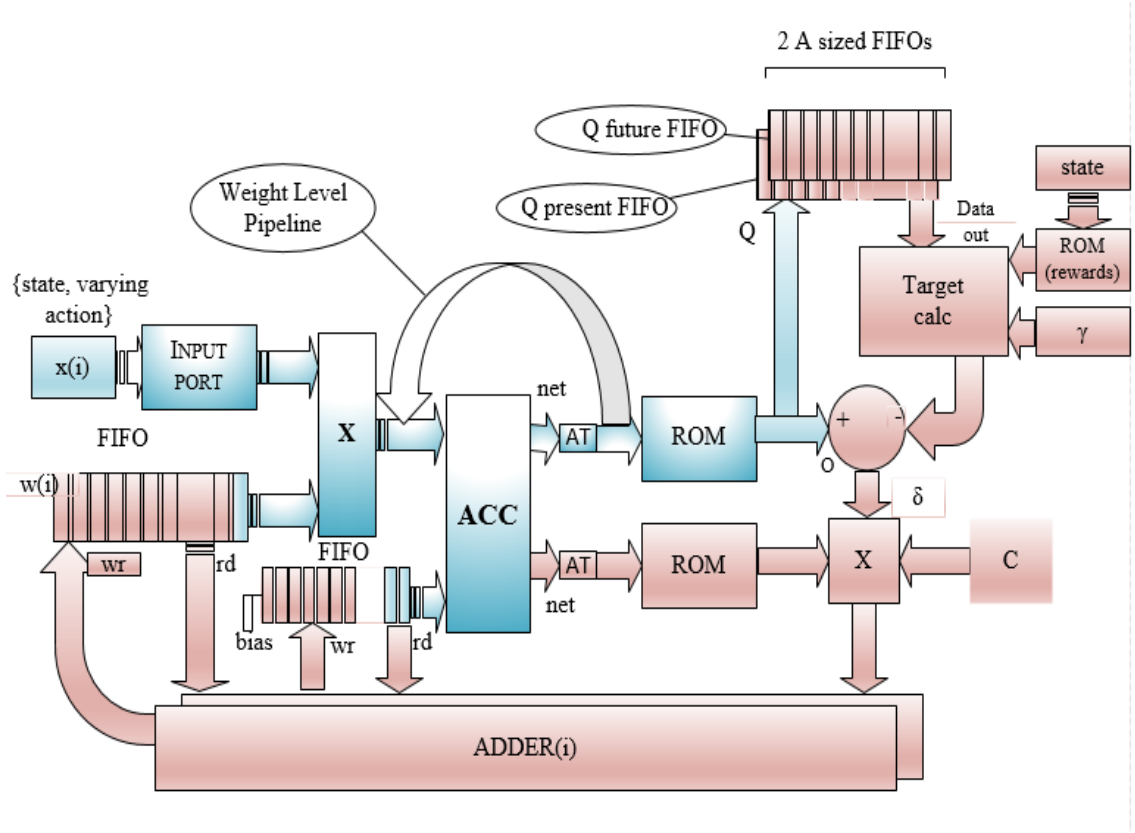

Figure 48. Weight Level Pipelining Implementation for Q-learning

A small sized ROM usage reduces the accuracy of the activation function calculation, instead piece wise linear activation function as demonstrated in [13] can be implemented for calculating various type of activation outputs. Action replay mechanism as demonstrated in [6] stores the experiences which are the states, actions and the utilities the agent has exploited. These experiences can be stored in a buffer memory which can further improve the process of training.

REFERENCES

[1] Watkins, C. J. C. H., & Dayan, P. (1992). Q-learning. Machine Learning, 8(3;3-4;), 279-292. doi:10.1023/A:1022676722315;10.1007/BF00992698;

[2] McGovern, A., & Wagstaff, K. L. (2011). Machine learning in space: Extending our reach. Machine Learning, 84(3), 335-340. doi:10.1007/s10994-011-5249-4

[3] Estlin, T. A., Bornstein, B. J., Gaines, D. M., Anderson, R. C., Thompson, D. R., Burl, M., . . . Judd, M. (2012). AEGIS automated science targeting for the MER opportunity rover. ACM Transactions on Intelligent Systems and Technology, 3(3) doi:10.1145/2168752.2168764

[4] Sutton, R. S., & Barto, A. G. (1998). Reinforcement learning: An introduction. Cambridge, Mass: MIT Press.

[5] Kober, J., Bagnell, J. A., & Peters, J. (2013). Reinforcement learning in robotics: A survey. The International Journal of Robotics Research, 32(11), 1238-1274. doi:10.1177/0278364913495721

[6] Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., & Riedmiller, M. (2013). Playing atari with deep reinforcement learning.

[7] Lake, B. M., Ullman, T. D., Tenenbaum, J. B., & Gershman, S. J. (2016). Building machines that learn and think like people.

[8] X. Peng, G. Berseth and M. van de Panne. Terrain-adaptive locomotion skills using deep reinforcement learning. ACM Transactions on Graphics (TOG) 35(4), pp. 1-12. 2016. . DOI: 10.1145/2897824.2925881.

[9] Levine, S., Wagener, N., & Abbeel, P. (2015). Learning contact-rich manipulation skills with guided policy search.

[10] Chen, X., & Lin, X. (2014). Big data deep learning: Challenges and perspectives. IEEE Access, 2, 514-525. doi:10.1109/ACCESS.2014.2325029

[11] Le, Q. V. (2013). Building high-level features using large scale unsupervised learning. Paper presented at the 8595-8598. doi:10.1109/ICASSP.2013.6639343

[12] Koomey, J. G. (2008). Worldwide electricity used in data centers. Environmental Research Letters, 3(3), 034008. doi:10.1088/1748-9326/3/3/034008

[13] Wang, C., Gong, L., Yu, Q., Li, X., Xie, Y., & Zhou, X. (2016). DLAU: A scalable deep learning accelerator unit on FPGA. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, , 1-1. doi:10.1109/TCAD.2016.2587683

[14] Zhang, C., Li, P., Sun, G., Guan, Y., Xiao, B., & Cong, J. (2015). Optimizing FPGA-based accelerator design for deep convolutional neural networks. Paper presented at the 161-170. doi:10.1145/2684746.2689060

[15] Liu, L., Luo, J., Deng, X., & Li, S. (2015). FPGA-based acceleration of deep neural networks using high level method. Paper presented at the 824-827. doi:10.1109/3PGCIC.2015.103

[16] Large-scale reconfigurable computing in a microsoft datacenter. (2014). Paper presented at the 1-38. doi:10.1109/HOTCHIPS.2014.7478819

[17] Tang, L., & Liu, Y. (2014;2013;). Adaptive neural network control of robot manipulator using reinforcement learning. Journal of Vibration and Control, 20(14), 2162-2171. doi:10.1177/1077546313482171

[18] Jeyanthi, S., & Subadra, M. (2014). Implementation of single neuron using various activation functions with FPGA. Paper presented at the 1126-1131. doi:10.1109/ICACCCT.2014.7019273

[19] Ormondi, A. R., Rajapakse, J. C., SpringerLink (Online service), & MyiLibrary. (2006). FPGA implementations of neural networks. Dordrecht, The Netherlands: Springer. doi:10.1007/0-387-28487-7

[20] Busoniu, L., & MyiLibrary. (2010). Reinforcement learning and dynamic programming using function approximators. Boca Raton, FL: CRC Press.

[21] Lacey, G., Taylor, G. W., & Areibi, S. (2016). Deep learning on FPGAs: Past, present, and future.

[22] Schwartz, H. M., & ebrary., I. (2014). Multi-agent machine learning: A reinforcement approach (1st ed.). Hoboken, New Jersey: John Wiley & Sons, Inc.

[23] Murugadoss, R., & Ramakrishnan, M. (2014). An effective performance of sigmoidal activation function in neural network architecture. International Journal of Applied Engineering Research, 9(22), 12097-12108.

[24] Glorot, X., & Bengio, Y. (2010). Understanding the difficulty of training deep feedforward neural networks. Journal of Machine Learning Research, 9, 249-256.

[25] Wang, Y., Yao, H., & Zhao, S. (2016;2015;). Auto-encoder based dimensionality reduction. Neurocomputing, 184, 232-242. doi:10.1016/j.neucom.2015.08.104

[26] Gysel, P., Motamedi, M., & Ghiasi, S. (2016). Hardware-oriented approximation of convolutional neural networks.

[27] Barry, B., Brick, C., Connor, F., Donohoe, D., Moloney, D., Richmond, R., . . . Toma, V. (2015). Always-on vision processing unit for mobile applications. IEEE Micro, 35(2), 56-66. doi:10.1109/MM.2015.10

[28] McMillan, R. (2016, May 19). WSJ.D technology: Google builds own high-speed chip --- tensor processing unit is 10 times faster than alternatives; the tech company says. Wall Street Journal

[29] Goldberg, L. (2016, April 7). 5 jaw-dropping things to catch at nvidia's GPU tech conference. Product Design & Development

[30] Hruska, J. (2015, March 2). Qualcomm's cognitive compute processors are coming to snapdragon 820. ExtremeTech.Com

[31] Vanderbauwhede, W., & Benkrid, K. (2014). High-performance computing using FPGAs. () doi:10.1007/978-1-4614-1791-0

[32] Wilson, P. (2011). Design recipes for FPGAs: Using verilog and VHDL. Burlington: Newnes.

[33] Maei, H. R., Szepesvari, C., Bhatnagar, S., & Sutton, R. S. (2010). Toward off-policy learning control with function approximation. Paper presented at the 719-726.

[34] Strehl, A. L., Li, L., Wiewiora, E., Langford, J., & Littman, M. L. (2006). PAC model-free reinforcement learning. Proceedings of the 23rd International Conference on Machine Learning - ICML '06. doi:10.1145/1143844.1143955.

[35] Altuntas, N., Imal, E., Emanet, N., & Ozturk, C. (2016). Reinforcement learning-based mobile robot navigation. Turkish Journal of Electrical Engineering and Computer Sciences, 24(3), 1747-1767. doi:10.3906/elk-1311-129

[36] Van Seijen, H., Van Hasselt, H., Whiteson, S., & Wiering, M. (2009). A Theoretical and Empirical Analysis of Expected SARSA. Paper presented at the 177-184. doi:10.1109/ADPRL.2009.4927542