**DEEP REINFORCEMENT LEARNING BASED LOAD BALANCING POLICY FOR MANAGING TRAFFIC IN DATACENTER ENVIROMENT**

MASTERS THESIS

***Submitted by***

BL.EN.P2CSE16003ASHWINI R DOKE

***In partial fulfilment of the requirements for the award of the degree of***

**MASTER OF TECHNOLOGY**

IN

“COMPUTER SCIENCE AND ENGINEERING”

****

AMRITA SCHOOL OF ENGINEERING, BENGALURU

AMRITA VISHWA VIDYAPEETHAM

**BENGALURU 560035**

July-2018

**AMRITA VISHWA VIDYAPEETHAM**

**AMRITA SCHOOL OF ENGINEERING, BENGALURU 560035**

****

**BONAFIDE CERTIFICATE**

This is to certify that the thesis entitled **“DEEP REINFORCEMENT LEARNING BASED LOAD BALANCING POLICY FOR MANAGING TRAFFIC IN DATACENTER ENVIROMENT”** submitted by **ASHWINI R DOKE (BL.EN.P2CSE16003)** in partial fulfilment of the requirements for the award of the degree of **Master of Technology** in **COMPUTER SCIENCE AND ENGINEERING** is a bonafide record of the work carried out under my guidance and supervision at Amrita School of Engineering, Bengaluru.

CHAIRPERSON

Dr. Amudha J.

Chairperson,

Dept. of Computer Science and Engineering,

Amrita School of Engineering, Bengaluru.

SUPERVISOR

Dr.K.Sangeeta.

Asso.Professor**,**

Dept. of Computer Science and Engineering,

Amrita School of Engineering, Bengaluru.

This project report was evaluated by us on ……......

**EVALUATORS**

EXAMINER 1

# EXAMINER 2

**ACKNOWLEDGEMENT**

First of all, my humble pranams at the lotus feet of **AMMA** who has been driving force and a source of inspiration for all educational activities and extracurricular activities at AMRITA SCHOOL OF ENGINEERING, BENGALURU. I wish to express my sincere and heartfelt gratitude to those whose timely guidance has made me accomplish this project.

I would like to express my sincere thanks to our Director **Br. Dhanraj Swamiji** for providing us with the best resources and infrastructure which helped us in successfully completing the project**.**

I am highly indebted to our Associate Dean **Dr. Rakesh S. G.,** for inspiring and providing me with the excellent facilities which helped me in completing the project work on time.

I would like to extend my sincere gratitude to **Dr. Amudha J.**, Chairperson , Dept. of Computer Science and Engineering for motivating us and for extending all the support and guidance towards the successful completion of the project.

My sincere thanks and gratitude to my project guide **Dr.K.Sangeeta. Asso.Professor**, Computer Science, School of Engineering, Bengaluru for extending all the support and guidance throughout and also towards the successful completion of the project.

I hereby express my special thanks and sincere gratitude to **G. Radhakrishnan.,** Asst Professor, Department of Computer Science and Engineering, Bengaluru for all his suggestions, timely help and much needed technical inputs.

I would also like to specially thank my **friends** who have constantly helped me both directly and indirectly in completing this project work. Last but not the least, I am deeply indebted to my **family** for all their sacrifices, constant support and co-operation without which I wouldn’t have been able to get this far.

**Place: BENGALURU ASHWINI R DOKE**

**Date: 24/4/2018 BL.EN.P2CSE16003**

**CHAPTER 1**

**INTRODUCTION**

The Load balancers in data center, manages uniform balance of incoming traffic (e.g., HTTP requests to web servers). The software load balancer at L4-L7 layers are often called application firewalls. Such firewall has richer visibility of the application data and application usage pattern (e.g. visibility to URL requested by web client). In real deployments, incoming web requests are load balanced with different hand-crafted load balancing policies [1]. These policies include Round Robin based, Source Based, Least Connections based, Least Time based, Destination based, Source and destination based and their variants. Often these policies are tuned to traffic pattern and network characteristic and are cheaper and easy to implement. But these policies do not achieve perfect load balancing due to distinct reasons [2]. e.g. either stickiness, (same source connection goes to same webserver)) is not achieved. or in case of host failure, you lose all the traffic while restoring it. Hence these hand-crafted policies are not optimal

**1.1 MOTIVATION**

Network traffic management in this era of bigdata is becoming a challenging task and to maintain them with human support is becoming more expensive We can address this challenge by applying deep reinforcement learning to a network load balancer which will be both time and cost effective [3]. Deep reinforcement learning learns and adjusts continuously with dynamic environment due to which we can optimize the performance of load balancer without human support . It has been stated in [4] that reinforcement learning (Q Learning) will improves the Quality of load balancer, and also Deep learning can be applied in order to achieve efficient load balancing [5].

**1.2 OBJECTIVES**

In this project Deep reinforcement learning algorithm like DQN (Deep Q Networks) which has good memory and replay quality has been applied for balancing network traffic in efficient way without human support. Our objectives are:

* Implement DQN (Deep Reinforcement algorithm) to balance datacenter traffic
* Comparison of Deep Reinforcement Learning policy with other handcrafted Policies- Round Robin

**CHAPTER 2**

**LITERATURE SURVEY**

In the following section we describe first the required terminologies for Load Balancing

**2.1** **Preliminaries**

**2.1.1** **Load Balancing**

Network Load balancing is a important constituent of highly-available infrastructures commonly used to improve the performance and reliability of web sites, applications, databases and other services by distributing the workload across multiple servers. Load balancer has Ability to handle volatile workloads and scale to millions of requests per second. In order to understand the significance of a load balancer a network without Load balancer is considered in Fig 2.1. Here, the user connects directly to the web server, at xyz.com. If this single web server is down, the user will no longer be able to access the website. In addition, if many users try to access the server simultaneously then the user may experience slow load times or may be unable to connect at all.

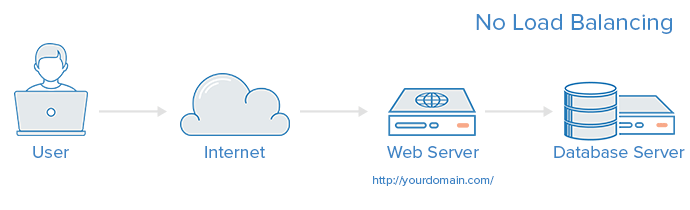


Fig 2.1: Network without Load Balancer.

This single point of failure can be mitigated by introducing a load balancer as shown in Fig 2.2 and at least one additional web server on the backend. Typically, all of the backend servers will supply identical content so that users receive consistent response from any one of the servers.. The user now, accesses the load balancer, which forwards the user's request to a backend server, which then responds directly to the user's request. In this scenario, the single point of failure is now the load balancer itself, which if required can be mitigated by introducing a second load balancer[5].

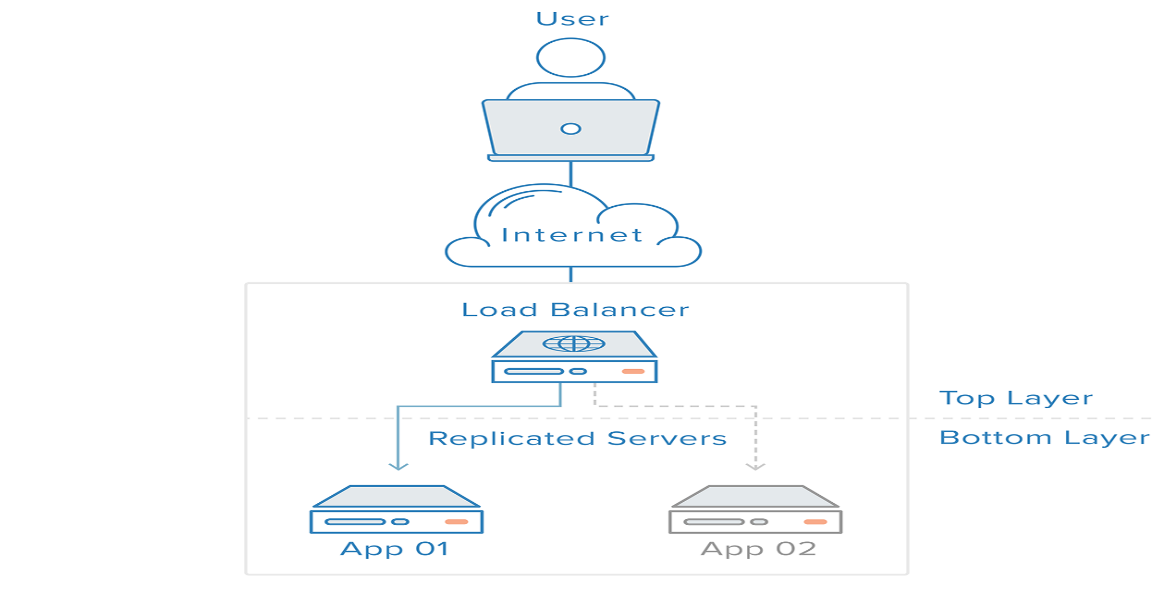


Fig 2.2: Network with Load Balancer.

**Load balancing** refers to efficiently distributing incoming network traffic across a group of backend servers, also known as a server farm or server pool. Modern high-traffic websites must serve hundreds of thousands, if not millions, of concurrent requests from users or clients and return the correct text, images, video, or application data, all in a fast and reliable manner. To cost-effectively scale to meet these high volumes, modern computing best practice generally requires adding more servers.

A [load balancer](https://www.nginx.com/solutions/load-balancing/) acts as the “traffic cop” sitting in front of your servers and routing client requests across all servers capable of fulfilling those requests in a manner that maximizes speed and capacity utilization and ensures that no one server is overworked, which could degrade performance. If a single server goes down, the load balancer redirects traffic to the remaining online servers. When a new server is added to the server group, the load balancer automatically starts to send requests to it.

Load balancer performs the following functions:

* Distributes client requests or network load efficiently across multiple servers
* Ensures high availability and reliability by sending requests only to servers that are online
* Provides the flexibility to add or subtract servers as demand dictates

The datacenter internet traffic is of different types and can be served by the fallowing two types of load balancer

* Forward Proxy load balancer

In deployment of Wide Area Network, network usage control and bandwidth optimization is important problem to solve. Such network architectures have thousands of clients machines tries to move content inside the WAN or they want to upload content outside the WAN. The important feature for such deployments is availability of internet service to client machines so they can connect to external webservers. Examples are, employees using internet for day to day browsing, checking emails on HTTP/HTTPS, being active on social media (Facebook/twitter etc.) or searching useful data for normal work or personal interests.

Therefore, important traffic in the WAN is HTTP/HTTPS based traffic which is originated in WAN and served from outside servers on which WAN administrator has little or no control.

It is therefore important to have such requests efficiently handled and load balanced properly and security and bandwidth policies should be applied. Most WAN architectures therefore deploy forward HTTP and HTTPS proxies which will do latter functionality. Such devices do HTTP/HTTPS caching, load balancing of requests, and supports connection persistence which helps for efficient and optimized traffic handling.

Proxies can operate at L4(TCP proxy) or it can be a L7(HTTP/FTP) proxy. Most of the time it also does SSL termination.

There are L3 layer load balancing techniques available as explained in [3] but they will not be discussed further.

A Layer 4-7 load balancer in a data center looks like below:

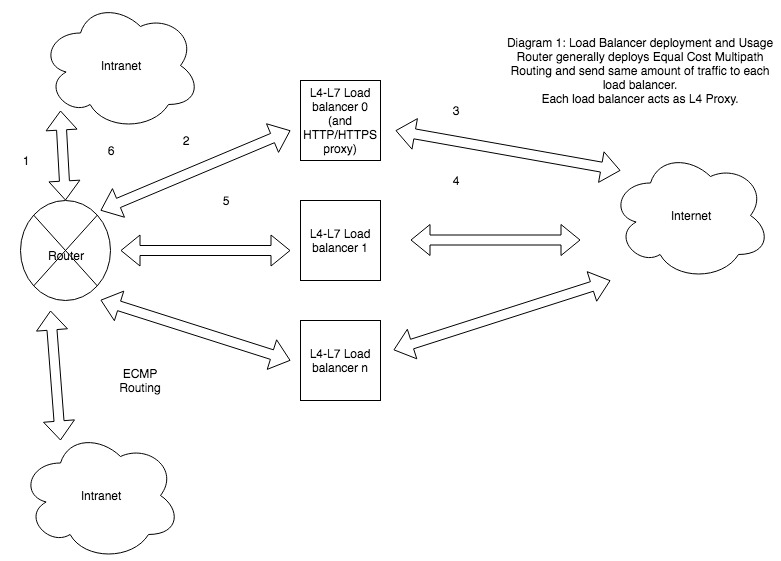


Fig 2.3 Forward Proxy

Here Intranet or multiple LAN’s are connected to Router. Router is responsible to steer application traffic (HTTP/HTTPS) to Load balancers and receiving response back from Load balancer and send it back to respective computer/node in LAN or intranet.

There is NAT and Firewall applied at intranet as well as between Load balancer and internet. But for simplicity we ignore it.

* **Reverse Proxy Load balancer**

The Reverse proxy in general serves one or multiple web servers. It receives all the requests on behalf of web server. It often does SSL termination for HTTPS traffic. It then load balances request to either of the web server.

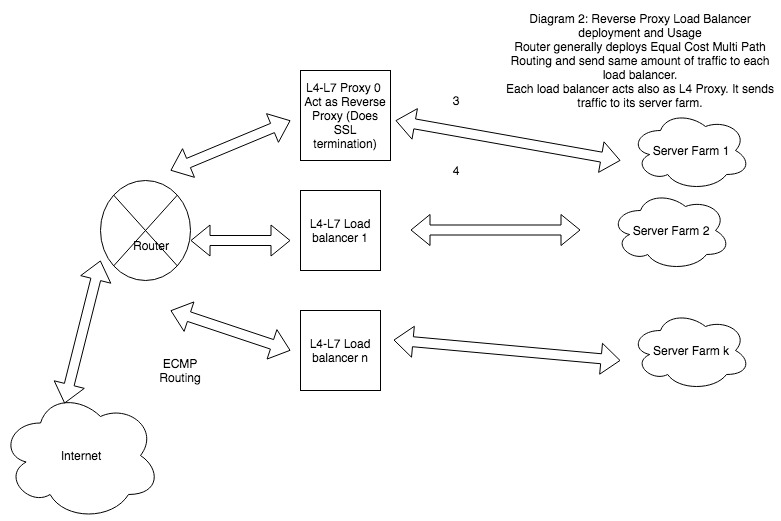


Fig 2.4 Reverse Proxy

**2.****1.2 Traffic Handled by Load Balancer.**

* **HTTP** — Standard HTTP balancing directs requests based on standard HTTP mechanisms. The Load Balancer sets the X-Forwarded-For, X-Forwarded-Proto, and X-Forwarded-Port headers to give the backends information about the original request.
* **HTTPS** — HTTPS balancing functions the same as HTTP balancing, with the addition of encryption. Encryption is handled in one of two ways: either with **SSL passthrough** which maintains encryption all the way to the backend or with **SSL termination** which places the decryption burden on the load balancer but sends the traffic unencrypted to the back end.
* **TCP** — For applications that do not use HTTP or HTTPS, TCP traffic can also be balanced. For example, traffic to a database cluster could be spread across all of the servers.
* **UDP** — More recently, some load balancers have added support for load balancing core internet protocols like DNS and syslog that use UDP.
  + 1. **Session Persistence**

An important issue when operating a load-balanced service is how to handle information that must be kept across the multiple requests in a user's session. If this information is stored locally on one backend server, then subsequent requests going to different backend servers would not be able to find it. This might be cached information that can be recomputed, in which case load-balancing a request to a different backend server just introduces a performance issue.

* + 1. **Load Balancing Policies**.
* Round Robin based:

Round robin DNS is often used to load balance requests between a number of Web servers. For example, a company has one domain name and three identical copies of the same web site residing on three servers with three different IP addresses. When one user accesses the home page it will be sent to the first IP address. The second user who accesses the home page will be sent to the next IP address, and the third user will be sent to the third IP address. In each case, once the IP address is given out, it goes to the end of the list. The fourth user, therefore, will be sent to the first IP address, and so forth.

If there are “n” takers or receptors for requests, each one will get a request to serve in turn. Its Easy to implement but Session Sticky will not be possible with this.

Round robin DNS load balancing works best for services with a large number of uniformly distributed connections to servers of equivalent capacity. Otherwise it just does [load distribution](https://en.wikipedia.org/wiki/Load_distribution).

* Source based (Client-Side load balancing):

Source based load balancing is also known as client side load balancing, It’s an

another approach to load balancing to deliver a list of server IPs to the client, and then to have client randomly select the IP from the list on each connection. This essentially relies on all clients generating similar loads, and the [Law of Large Numbers](https://en.wikipedia.org/wiki/Law_of_Large_Numbers)[[4]](https://en.wikipedia.org/wiki/Load_balancing_(computing)#cite_note-ithare-4) to achieve a reasonably flat load distribution across servers. It has been claimed that client-side random load balancing tends to provide better load distribution than round-robin DNS; this has been attributed to caching issues with round-robin DNS, that in case of large DNS caching servers, tend to skew the distribution for round-robin DNS, while client-side random selection remains unaffected regardless of DNS caching.[[4]](https://en.wikipedia.org/wiki/Load_balancing_(computing)#cite_note-ithare-4)

With this approach, the method of delivery of list of IPs to the client can vary, and may be implemented as a DNS list (delivered to all the clients without any round-robin), or via hardcoding it to the list. If a "smart client" is used, detecting that randomly selected server is down and connecting randomly again, it also provides fault tolerance.

The source IP or username is hashed and will be always sent to one of the service node to process. Source stickiness is possible. But destination stickiness is not achieved with this.

* Destination based (Server-Side Load Balancer):

A server-side load balancer is usually a software program that is listening on the [port](https://en.wikipedia.org/wiki/TCP_and_UDP_port) where external clients connect to access services. The load balancer forwards requests to one of the "backend" servers, which usually replies to the load balancer. This allows the load balancer to reply to the client without the client ever knowing about the internal separation of functions. It also prevents clients from contacting back-end servers directly, which may have security benefits by hiding the structure of the internal network and preventing attacks on the kernel's network stack or unrelated services running on other ports.

Some load balancers provide a mechanism for doing something special in the event that all backend servers are unavailable. This might include forwarding to a backup load balancer or displaying a message regarding the outage.

It is also important that the load balancer itself does not become a [single point of failure](https://en.wikipedia.org/wiki/Single_point_of_failure). Usually load balancers are implemented in [high-availability](https://en.wikipedia.org/wiki/High_availability) pairs which may also replicate session persistence data if required by the specific application.

The destination IP, server weight, or one of the header from HTTP request determines the value to be hashed and destination is determined from this hash. Destination based sticky connections are possible. If source connection is authenticated by load balancer then sticky connection for source side is not possible with this solution.

* Least Connections:

Least Connections based policy take the current server load into consideration when distributing requests. The current request goes to the server that is servicing the least number of active sessions at the current time.

The following example shows how to provide load balancing by using the least connections method. Consider the following three services:

Server-1 is handling 3 active transactions.

Server-2 is handling 15 active transactions.

Server-3 is not handling any active transactions.

The load balancer selects the service by using the value (N) of the following expression:  
N = Number of active transactions

The requests are delivered as follows:

Server-3 receives the first request because the service is not handling any active transactions.  
Note: The service with no active transaction is selected first.

Server-3 receives the second and third requests because the service has the next least number of active transactions.

Server-1 receives the fourth request.

When Server-1 and Server-3 have same number of active transactions, NetScaler performs load balancing in a round robin manner. Therefore, Server-3 receives the fifth request, Server-1 receives the sixth request, Server-3 receives the seventh request, and Server-1 receives the eighth request and so forth. But with this policy stickiness is not possible to achieve.

**2.1.5.** **Health Checks**

Load balancers should only forward traffic to "healthy" backend servers. To monitor the health of a backend server, health checks regularly attempt to connect to backend servers using the protocol and port defined by the forwarding rules to ensure that servers are listening. If a server fails a health check, and therefore is unable to serve requests, it is automatically removed from the pool, and traffic will not be forwarded to it until it responds to the health checks again.

**2.2. Reinforcement Learning**

Reinforcement Learning (RL) is an active area of research in AI because of its widespread applicability in both accessible and inaccessible environments. The model of the reinforcement learning problem is based on the theory of Markov Decision Processes (MDP) (Stone and Veloso, 1997).

In RL, an agent learns by interacting with its environment and tries to maximize its long term return by performing actions and receiving rewards as shown in Fig 2.3. This area of machine learning learns the behavior of dynamic environment through trial and error. The trial and error learning feature and the concept of reward makes the reinforcement learning distinct from other learning techniques.

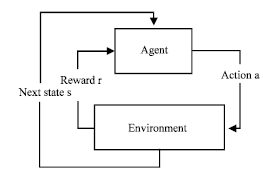


Fig 2.5 Reinforcement Learning

Reinforcement learning in general and Q-learning in particular can be applied to dynamic load balancing and scheduling in distributed heterogeneous system. And it has been proved that Q-learning improves the quality of load balancing in large scale heterogeneous systems[6].

* **Q Learning Algorithm Invention**

In 1950 American mathematician Richard Bellman tring to solve Optimal Control Problem Optimal Control Problem: Designing of an agent to minimize some behavior of a System over time Solution: Bellman Equation: Describes the value of a problem at a certain point of time in terms of previous decisions Application: Minimizing Flight Time for airplane.

After few time Phycologist- Edward Thorndike tring to understand how Learning works so he invented Low Of Effect “Responses That Produce satisfying effect in a particular situation become more likely to Occur again in that situation & Responses That Produce Dissatisfying effect in a particular situation become less likely to occur again in that situation “one of his .Experiment: Putting a cat in a box and observing it while it tries to bunch of different ways to come out of the box Finally hit the lever that open the box Again he put a cat in Box then he immediately come to know how to come out By process of trial and error

Finally, British computer Scientist name Chris Watkins Combines two ideas one is Bellman Equation& process of. Trial and Error Process (Low of Effect) and he invented Q Learning

The Q-Learning algorithm was proposed as a way to optimize solutions in Markov decision process problems.  The distinctive feature of Q-Learning is in its capacity to choose between immediate rewards and delayed rewards.  At each step of time, an agent observes the vector of state xt, then chooses and applies an action ut. As the process moves to state xt+1, the agent receives a reinforcement r(xt, ut).  The goal of the training is to find the sequential order of actions which maximizes the sum of the future reinforcements, thus leading to the shortest path from start to finish.

The transition rule of Q learning :

Q(state, action) = R(state, action) + gamma \* Max[Q(next state, all actions)]

The gamma parameter has a range of 0 to 1 (0 <= gamma > 1), and ensures the convergence of the sum.  If gamma is closer to zero, the agent will tend to consider only immediate rewards.  If gamma is closer to one, the agent will consider future rewards with greater weight, willing to delay the reward.

The Q-Learning algorithm goes as follows:

1. Set the gamma parameter, and environment rewards in matrix R.

2. Initialize matrix Q to zero.

3. For each episode:

Select a random initial state.

Do While the goal state hasn't been reached.

Select one among all possible actions for the current state.

Using this possible action, consider going to the next state.

Get maximum Q value for this next state based on all possible actions.

Compute: Q(state, action) = R(state, action) + gamma \* Max[Q(next state, all actions)]

Set the next state as the current state.

End Do

End For

Example of Q Learning.

Suppose we have 5 rooms in a building connected by doors as shown in the figure below.  We'll number each room 0 through 4.  The outside of the building can be thought of as one big room (5).  Notice that doors 1 and 4 lead into the building from room 5 (outside).

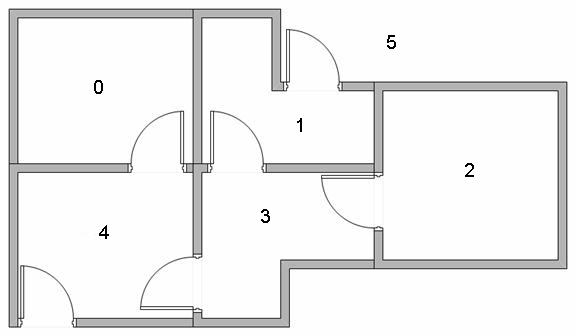


Fig 2.6 Example of Q Learning

We can represent the rooms on a graph, each room as a node, and each door as a link.

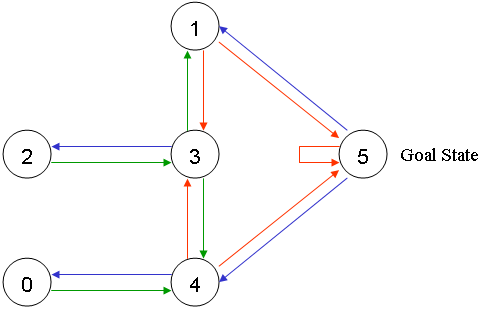


Fig 2.7 Graphical representation of Q learning Example

For this example, we'd like to put an agent in any room, and from that room, go outside the building (this will be our target room). In other words, the goal room is number 5. To set this room as a goal, we'll associate a reward value to each door (i.e. link between nodes). The doors that lead immediately to the goal have an instant reward of 100.  Other doors not directly connected to the target room have zero reward. Because doors are two-way ( 0 leads to 4, and 4 leads back to 0 ), two arrows are assigned to each room. Each arrow contains an instant reward value, as shown below:

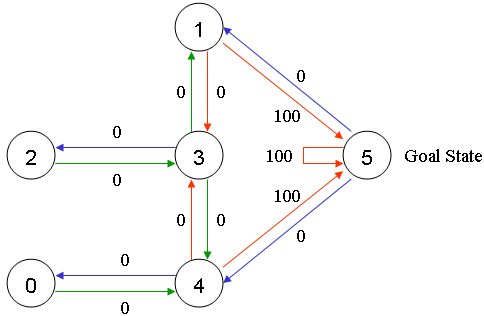


Fig 2.8 Explanation of Q Learning

Room 5 loops back to itself with a reward of 100, and all other direct connections to the goal room carry a reward of 100.  In Q-learning, the goal is to reach the state with the highest reward, so that if the agent arrives at the goal, it will remain there forever. This type of goal is called an "absorbing goal".

Imagine our agent as a dumb virtual robot that can learn through experience. The agent can pass from one room to another but has no knowledge of the environment, and doesn't know which sequence of doors lead to the outside.

Suppose we want to model some kind of simple evacuation of an agent from any room in the building. Now suppose we have an agent in Room 2 and we want the agent to learn to reach outside the house (5).

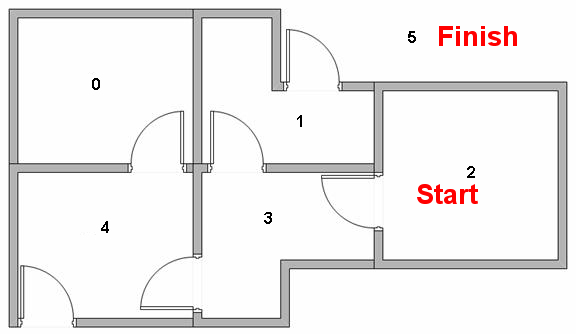


Fig 2.9 State of agent and action taken.

The terminology in Q-Learning includes the terms "state" and "action".

We'll call each room, including outside, a "state", and the agent's movement from one room to another will be an "action".  In our diagram, a "state" is depicted as a node, while "action" is represented by the arrows.

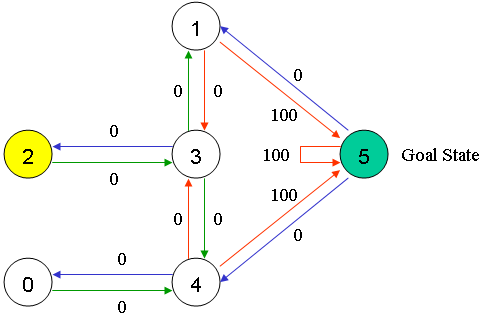
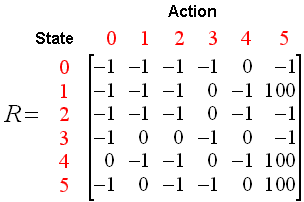


Fig 2.10 Q Learning Example

Suppose the agent is in state 2.  From state 2, it can go to state 3 because state 2 is connected to 3.  From state 2, however, the agent cannot directly go to state 1 because there is no direct door connecting room 1 and 2 (thus, no arrows).  From state 3, it can go either to state 1 or 4 or back to 2 (look at all the arrows about state 3).  If the agent is in state 4, then the three possible actions are to go to state 0, 5 or 3.  If the agent is in state 1, it can go either to state 5 or 3.  From state 0, it can only go back to state 4.

We can put the state diagram and the instant reward values into the following reward table, "matrix R".



The -1's in the table represent null values (i.e.; where there isn't a link between nodes). For example, State 0 cannot go to State 1.

Now we'll add a similar matrix, "Q", to the brain of our agent, representing the memory of what the agent has learned through experience.  The rows of matrix Q represent the current state of the agent, and the columns represent the possible actions leading to the next state (the links between the nodes).

The agent starts out knowing nothing, the matrix Q is initialized to zero.  In this example, for the simplicity of explanation, we assume the number of states is known (to be six).  If we didn't know how many states were involved, the matrix Q could start out with only one element.  It is a simple task to add more columns and rows in matrix Q if a new state is found.

The transition rule of Q learning is a very simple formula:

Q(state, action) = R(state, action) + Gamma \* Max[Q(next state, all actions)]

According to this formula, a value assigned to a specific element of matrix Q, is equal to the sum of the corresponding value in matrix R and the learning parameter Gamma, multiplied by the maximum value of Q for all possible actions in the next state.

Our virtual agent will learn through experience, without a teacher (this is called unsupervised learning).  The agent will explore from state to state until it reaches the goal. We'll call each exploration an episode.  Each episode consists of the agent moving from the initial state to the goal state.  Each time the agent arrives at the goal state, the program goes to the next episode.

The Q-Learning algorithm goes as follows:

1. Set the gamma parameter, and environment rewards in matrix R.

2. Initialize matrix Q to zero.

3. For each episode:

Select a random initial state.

Do While the goal state hasn't been reached.

Select one among all possible actions for the current state.

Using this possible action, consider going to the next state.

Get maximum Q value for this next state based on all possible actions.

Compute: Q(state, action) = R(state, action) + Gamma \* Max[Q(next state, all actions)]

Set the next state as the current state.

End Do

End For

The algorithm above is used by the agent to learn from experience.  Each episode is equivalent to one training session.  In each training session, the agent explores the environment (represented by matrix R ), receives the reward (if any) until it reaches the goal state. The purpose of the training is to enhance the 'brain' of our agent, represented by matrix Q.  More training results in a more optimized matrix Q.  In this case, if the matrix Q has been enhanced, instead of exploring around, and going back and forth to the same rooms, the agent will find the fastest route to the goal state.

The Gamma parameter has a range of 0 to 1 (0 <= Gamma > 1).  If Gamma is closer to zero, the agent will tend to consider only immediate rewards.  If Gamma is closer to one, the agent will consider future rewards with greater weight, willing to delay the reward.

To use the matrix Q, the agent simply traces the sequence of states, from the initial state to goal state.  The algorithm finds the actions with the highest reward values recorded in matrix Q for current state:

Algorithm to utilize the Q matrix:

1. Set current state = initial state.

2. From current state, find the action with the highest Q value.

3. Set current state = next state.

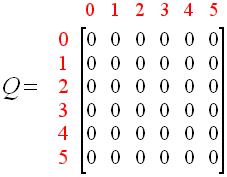
4. Repeat Steps 2 and 3 until current state = goal state.

The algorithm above will return the sequence of states from the initial state to the goal state.

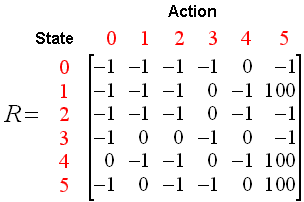
Mathematical Explaination of Q learning.

Lets consider  learning parameter Gamma = 0.8, and the initial state as Room 1.

Initialize matrix Q as a zero matrix:



Look at the second row (state 1) of matrix R.  There are two possible actions for the current state 1: go to state 3, or go to state 5. By random selection, we select to go to 5 as our action.



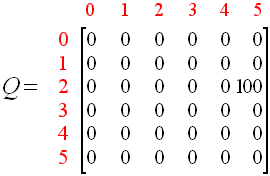
Now let's imagine what would happen if our agent were in state 5.  Look at the sixth row of the reward matrix R (i.e. state 5).  It has 3 possible actions: go to state 1, 4 or 5.

Q(state, action) = R(state, action) + Gamma \* Max[Q(next state, all actions)]

Q(1, 5) = R(1, 5) + 0.8 \* Max[Q(5, 1), Q(5, 4), Q(5, 5)] = 100 + 0.8 \* 0 = 100

Since matrix Q is still initialized to zero, Q(5, 1), Q(5, 4), Q(5, 5), are all zero.  The result of this computation for Q(1, 5) is 100 because of the instant reward from R(5, 1).

The next state, 5, now becomes the current state.  Because 5 is the goal state, we've finished one episode.  Our agent's brain now contains an updated matrix Q as:



For the next episode, we start with a randomly chosen initial state.  This time, we have state 3 as our initial state.

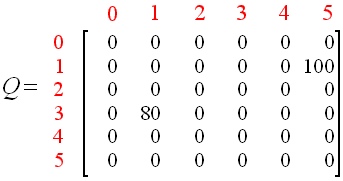
Look at the fourth row of matrix R; it has 3 possible actions: go to state 1, 2 or 4.  By random selection, we select to go to state 1 as our action.

Now we imagine that we are in state 1.  Look at the second row of reward matrix R (i.e. state 1).  It has 2 possible actions: go to state 3 or state 5.  Then, we compute the Q value:

Q(state, action) = R(state, action) + Gamma \* Max[Q(next state, all actions)]

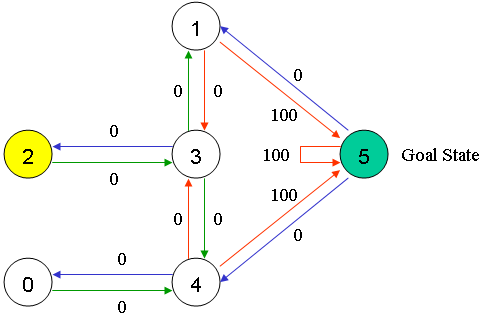
Q(1, 5) = R(1, 5) + 0.8 \* Max[Q(1, 2), Q(1, 5)] = 0 + 0.8 \* Max(0, 100) = 80

We use the updated matrix Q from the last episode.  Q(1, 3) = 0 and Q(1, 5) = 100.  The result of the computation is Q(3, 1) = 80 because the reward is zero.  The matrix Q becomes:



The next state, 1, now becomes the current state.  We repeat the inner loop of the Q learning algorithm because state 1 is not the goal state.

So, starting the new loop with the current state 1, there are two possible actions: go to state 3, or go to state 5.  By lucky draw, our action selected is 5.



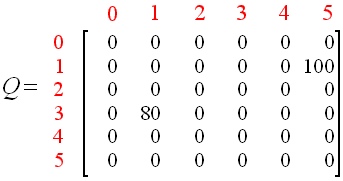
Now, imaging we're in state 5, there are three possible actions: go to state 1, 4 or 5.  We compute the Q value using the maximum value of these possible actions.

Q(state, action) = R(state, action) + Gamma \* Max[Q(next state, all actions)]

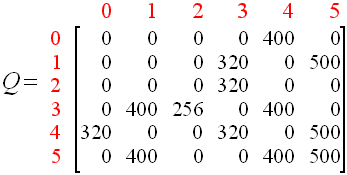
Q(1, 5) = R(1, 5) + 0.8 \* Max[Q(5, 1), Q(5, 4), Q(5, 5)] = 100 + 0.8 \* 0 = 100

The updated entries of matrix Q, Q(5, 1), Q(5, 4), Q(5, 5), are all zero.  The result of this computation for Q(1, 5) is 100 because of the instant reward from R(5, 1).  This result does not change the Q matrix.

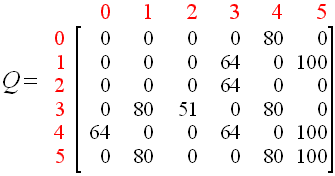
Because 5 is the goal state, we finish this episode.  Our agent's brain now contain updated matrix Q as:



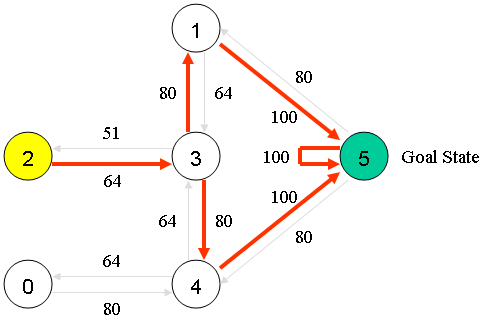
If our agent learns more through further episodes, it will finally reach convergence values in matrix Q like:



This matrix Q, can then be normalized (i.e.; converted to percentage) by dividing all non-zero entries by the highest number (500 in this case):



Once the matrix Q gets close enough to a state of convergence, we know our agent has learned the most optimal paths to the goal state.  Tracing the best sequences of states is as simple as following the links with the highest values at each state.



For example, from initial State 2, the agent can use the matrix Q as a guide:

From State 2 the maximum Q values suggests the action to go to state 3.

From State 3 the maximum Q values suggest two alternatives: go to state 1 or 4.  Suppose we arbitrarily choose  to go to 1.

From State 1 the maximum Q values suggests the action to go to state 5.

Thus the sequence is 2 - 3 - 1 - 5.

**2.3 Deep Reinforcement Learning**

Humans excel at solving a wide variety of challenging problems, from low-level motor control through to high-level cognitive tasks.

Deep Reinforcement learning creates artificial agents that can achieve a similar level of performance and generality. Like a human, our agents learn for themselves to achieve successful strategies that lead to the greatest long-term rewards.

This paradigm of learning by trial-and-error, solely from rewards or punishments, is known as [reinforcement learning](https://en.wikipedia.org/wiki/Reinforcement_learning) (RL). Also like a human, our agents construct and learn their own knowledge directly from raw inputs, such as vision, without any hand-engineered features or domain heuristics. This is achieved by [deep learning](https://en.wikipedia.org/wiki/Deep_learning) of neural networks

Deep Reinforcement learning seek a single agent which can solve any human-level task. where Reinforcement Learning(RL) defines an objective Deep Learning(DL) gives the mechanism to achieve that objective. Which will result into general intelligence.

It has been proved that deep learning[7] model can be used to successfully learn control policies directly from high-dimensional sensory input using reinforcement learning. The model is a convolutional neural network, trained with a variant of Q-learning, whose input is raw pixels and whose output is a value function estimating future rewards. and

Deep Learning has been applied to seven Atari 2600 games from the Arcade Learn- ing Environment, with no adjustment of the architecture or learning algorithm. It has been found that it outperforms all previous approaches on six of the games and surpasses a human expert on three of them[8].

Application: Automatic Strategy selection e.g. Play & Win Atari games.

**2.4 Deep Reinforcement Learning Algorithms**

 Deep reinforcement learning algorithms are used to create the first artificial agents to achieve human-level performance across many challenging domains. Such agents learn for themselves to achieve successful strategies that lead to the greatest long-term rewards. This paradigm of learning by trial-and-error, solely from rewards or punishments, is known as [reinforcement learning](https://en.wikipedia.org/wiki/Reinforcement_learning) (RL). Also like a human, agents construct and learn their own knowledge directly from raw inputs, such as vision, without any hand-engineered features or domain heuristics. This is achieved by [deep learning](https://en.wikipedia.org/wiki/Deep_learning) of neural networks. At Deep Mind (Googles Company) Combines These two approaches to develop Deep reinforcement learning Algorithms.fallowing are the common Deep Reinforcement learning algorithms.

**2.4.1** **Deep Q Network (DQN))**

In DQNAgents must continually make value judgements so as to select good actions over bad. This knowledge is represented by a Q-network that estimates the total reward that an agent can expect to receive after taking a particular action. Two years ago Deep Mind introduced the first widely successful [algorithm](http://arxiv.org/pdf/1312.5602.pdf) for deep reinforcement learning. The key idea was to use deep neural networks to represent the Q-network, and to train this Q-network to predict total reward. Previous attempts to combine RL with neural networks had largely failed due to unstable learning. To address these instabilities, our Deep Q-Networks (DQN) algorithm stores all of the agent's experiences and then randomly samples and replays these experiences to provide diverse and decorrelated training data. DQN is a variant of the Q-learning algorithm (Watkins, 1989) that utilizes deep learning. It uses a neural network to parametrize an approximation for the action-value function Q(s, a; θ) using parameters θ.

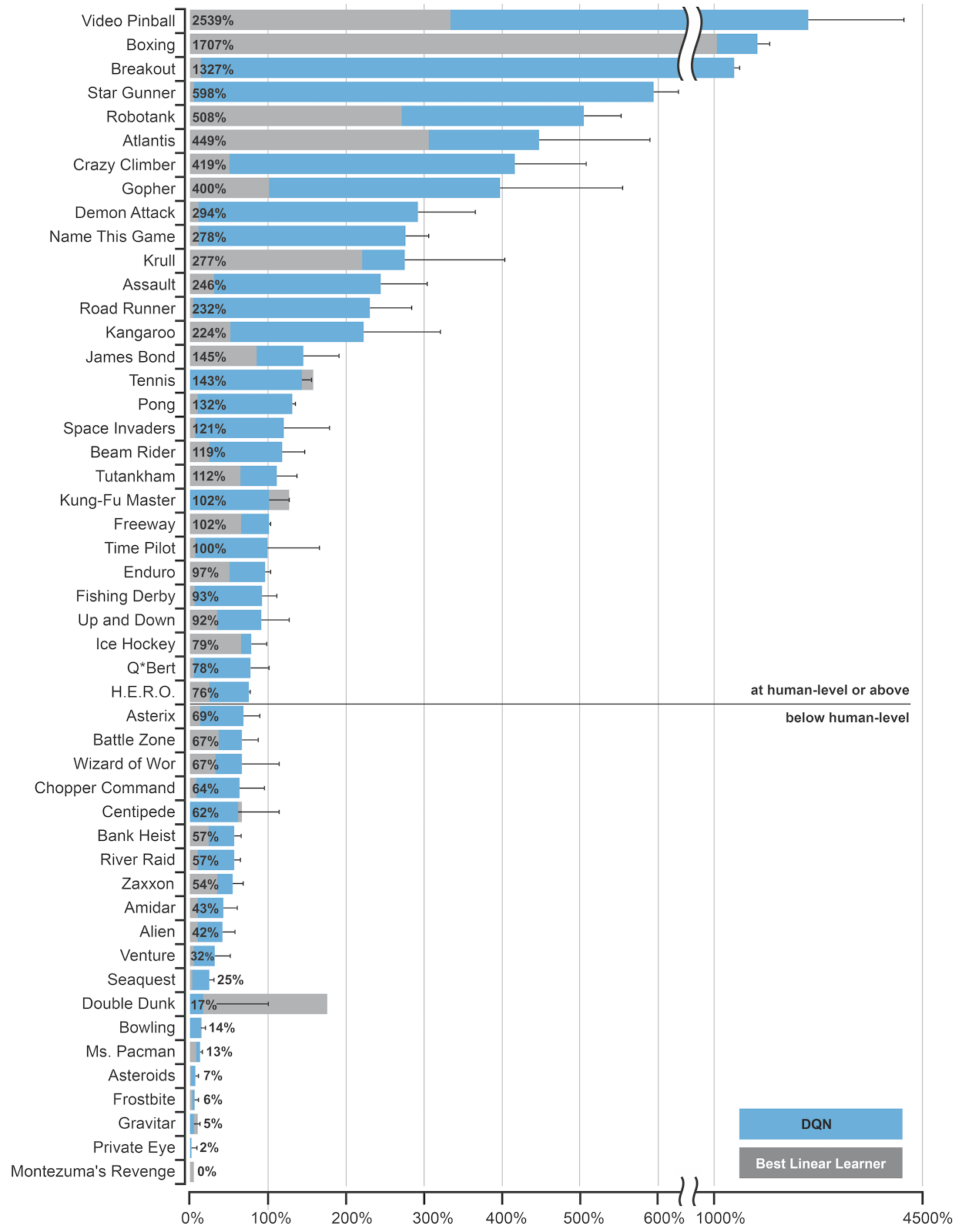
Deep Mind applied DQN to learn to play games on the Atari 2600 console. At each time-step the agent observes the raw pixels on the screen, a reward signal corresponding to the game score, and selects a joystick direction. In its [Nature paper](https://storage.googleapis.com/deepmind-data/assets/papers/DeepMindNature14236Paper.pdf) deep mind trained separate DQN agents for 50 different Atari games, without any prior knowledge of the game rules. And it has been proved that DQN achieved human-level performance in almost half of the 50 games to which it was applied; far beyond any previousmethod. 

Fig 2.11 Performance of DQN

**2.4.2 Asynchronous Advantage Actor-Critic** (A3C)

A3C[9] exploits the multithreading capabilities of standard CPUs. The idea is to execute many instances of our agent in parallel but using a shared model. This provides a viable alternative to experience replay, since parallelization also diversifies and decorrelates the data. Deep minds asynchronous actor-critic algorithm, [A3C](http://arxiv.org/pdf/1602.01783), combines a deep Q-network with a deep policy network for selecting actions. It achieves state-of-the-art results, using a fraction of the training time of DQN and a fraction of the resource consumption of Gorila. By building novel approaches to [intrinsic motivation](https://arxiv.org/abs/1606.01868) and [temporally abstract planning](http://arxiv.org/pdf/1606.04695), Deep Mind also achieved breakthrough results in the most notoriously challenging Atari games, such as Montezuma’s Revenge.

Asynchronous implementation of the advantage actor-critic paradigm where separate threads run in parallel and perform updates to shared parameters. The different threads each hold their own instance of the environment and have different exploration policies, thereby decorrelating parameter updates without the need for experience replay. Therefore, A3C is an online algorithm, whereas DQN learns its policy offline,

**2.4.3 DARLA ((Disent Angled Representation Learning Agent)**

DARLA[11] basically used to Improv Zero-Shot Transfer in Reinforcement Learning. Domain adaptation is an important open problem in deep reinforcement learning (RL). In many scenarios of interest data is hard to obtain, so agents may learn a source policy in a setting where data is readily available, with the hope that it generalizes well to the target domain.

It’s a new multi-stage RL agent, DARLA (Disent Angled Representation Learning Agent), which learns to see before learning to act. DARLA’s vision is based on learning a disentangled representation of the observed environment. Once DARLA can see, it is able to acquire source policies that are robust to many domain shifts - even with no access to the target domain. DARLA significantly outperforms conventional baselines in zero-shot domain adaptation scenarios, an effect that holds across a variety of RL environments and base RL algorithms DQN, A3C

**2.5 Related work**

Reinforcement learning learns the behavior of dynamic environment through trial and error & it has been proved that Q learning improves the Quality of load balancer [4], & Computer systems can optimize their own performance by learning from experience without human assistance. To repeatedly adjust in response to a dynamic environment, they will need the adaptability that only machine learning can offer. In this regard, the use of Reinforcement Learning is more precise and potentially computationally cheaper than other approaches.[4]

Deep learning can be applied in order to achieve efficient load balancing . As large number of the event data will be generated over a period of time in IoT. Hence, the load balancing protocol is critical considerations in the design of IoT. Therefore, if we propose an agent Loadbot that measures network load and process structural configuration by analyzing a large amount network load, and applying Deep Learning’s Deep Belief Network method in order to achieve efficient load balancing [11]

Load Balancer with Artificial Neural Network[12]discusses a proposed load balance technique based on artificial neural network. It distributes workload equally across all the nodes by using back propagation learning algorithm to train feed forward Artificial Neural Network (ANN). The proposed technique is simple and it can work efficiently when effective training sets are used. ANN predicts the demand and thus allocates resources according to that demand. Thus, it always maintains the active servers according to current demand, which results in low energy consumption than the conservative approach of over-provisioning. Furthermore, high utilization of server results in more power consumption, server running at higher utilization can process more workload with similar power usage. Finally the existing load balancing techniques in cloud computing are discussed and compared with the proposed technique based on various parameters like performance, scalability, associated overhead... etc. In addition energy consumption and carbon emission perspective are also considered to satisfy green computing.

In this project we are going to apply Deep reinforcement learning algorithm like DQN (Deep Q Networks) which has good memory and replay quality[13] and so can be applied for balancing network traffic in efficient way without human support.

**CHAPTER 3**

**SYSTEM REQUIREMENT**

System requirements or configuration that a system must have in order to achieve effective load Balancing are as below

**3.1 Software Requirement**

* Operating System -Mac OS X Sierra.
* Software - Python version 3.6.
* Operating System - Mac OS X Sierra.
* Python Modules - redis, TensorFlow, Keras

**3.2** **Hardware Requirement**.

* Processor: Intel quad core I7
* RAM – 16 GB
* Hard disk – 100 GB

**CHAPTER 4**

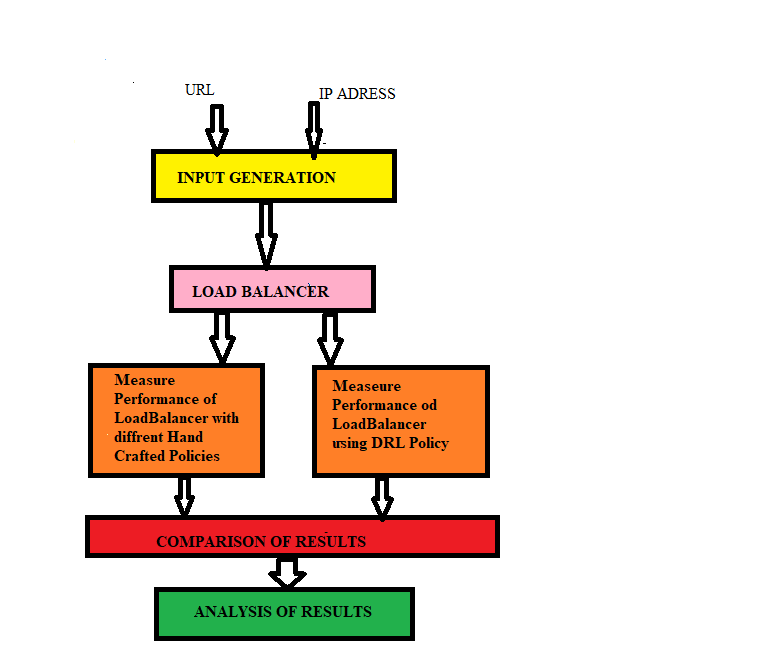
**SYSTEM ARCHITECTURE**

**Load balancing** refers to efficiently distributing incoming network traffic across a group of backend servers, also known as a server farm or server pool. load balancer performs the following functions:

* Distributes client requests or network load efficiently across multiple servers
* Ensures high availability and reliability by sending requests only to servers that are online
* Provides the flexibility to add or subtract servers as demand dictates

In Order to perform above functions efficiently our System has to be modeled as Shown in Fig.4.1.

* Combination of top 1 million URL and 2000 ,30 Bit IP Addresses are used to generate 10 million inputs.
* Input is given to Load Balancer
* Measure Performance of Load Balancer with Respect to DRL Policy and Hand crafted Policies.
* Comparison of Both approaches
* Analysis of Results.



**Fig 4.1 System Architecture**

**CHAPTER 5**

**IMPLEMENTATION DETAILS**

**5.1. Experimental Setup**

Fig 5.1. Shows the illustrative arrangement of load balancing scheme in an enterprise data center which consisting of a set of LANs connected together, Load balancer will try to send traffic to one of many proxy servers such that, all internet access requests will be fairly served by the devices. There can be multiple such load balancers and forward proxies.

The forward proxies will serve a set of host from the datacenter at one end and connect them to internet based origin server at other end . Internet based origin servers represent a distributed and loosely coupled architecture. and provide dynamic or static content based on geo-location/authenticated-user/IP/user-preferences.

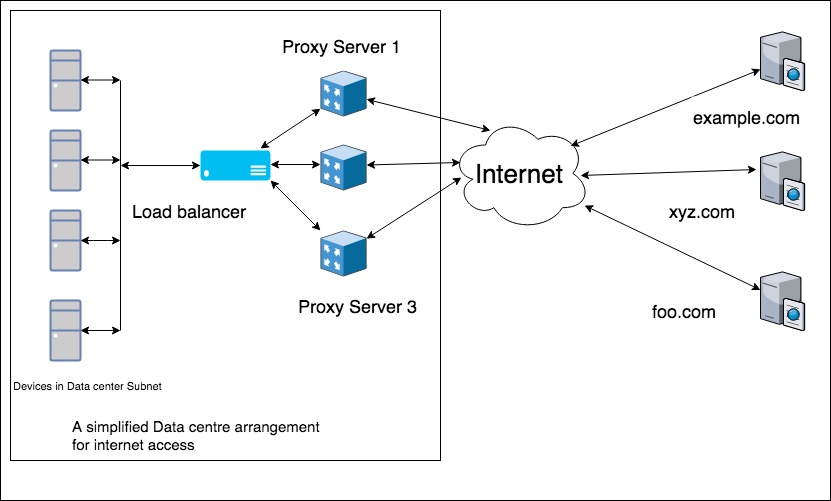


Fig 5.1 Load balancing Forward proxies in Data centre.

The Input and Output generated is described below:

**5.2. Input and Output Generation**

[input]

min\_mask = 30

max\_ip = 2000

ip\_address = s\_ip.txt

url\_file = top-1m.csv

max\_distributions = 5

; 10 million times.

iterations = 10000000

experiement\_input\_file = exp\_input.txt

[distribution0]

start = 0

end = 99

weight = 35

[distribution1]

start = 100

end = 999

weight = 30

[distribution2]

start = 1000

end = 9999

weight = 20

[distribution3]

start = 10000

end = 99999

weight = 10

[distribution4]

start = 100000

end = 999999

weight = 5

[output]

http\_persistent\_connection\_size = 200000

http\_persistent\_timeout = 300

persistent\_hit\_award = 1

persistent\_miss\_award = 0

policies are round\_robin, least\_connection, source\_hash, destination\_hash, source\_and\_destination\_hash, reinforcement, deep\_reinforcement,

policy = deep\_reinforcement

redis\_servers\_ports = 6379, 6381, 6383

total\_servers = 3

download\_time = 1, 2, 3, 5, 10, 15, 20, 25, 30, 60, 300

download\_time\_weight = 30, 25, 20, 12, 6, 2, 1, 1, 1, 1, 1

persistent\_timeout = 300

report\_file = drl2.csv

max\_iteration = 5000

**5.3 Algorithm**

In Q-Learning Algorithm, there is a function called Q Function, which is used to approximate the reward based on a state. We call it Q(s,a), where Q is a function which calculates the expected future value from state s and action a. Similarly in Deep Q Network algorithm, we use a neural network to approximate the reward based on the state. With this notifications the deep Q Learning algorithm is as fallow

Algorithm,

1. Set the gamma parameter, and environment rewards

2. Initialize matrix Q to zero.

3. For each episode:

Select a random initial state.

Do While the goal state hasn't been reached.

Select one among all possible actions for the current state.

Using this possible action, consider going to the next state.

Get maximum Q value for this next state based on all possible actions.

Compute: Q(state, action) = R(state, action) + Gamma \* Max[Q(next state, all actions)]

Set the next state as the current state.

End Do

End For

The algorithm above is used by the agent to learn from experience.  Each episode is equivalent to one training session.  In each training session, the agent explores the environment (represented by matrix R ), receives the reward (if any) until it reaches the goal state.

Q(state, action) = R(state, action) + Gamma \* Max[Q(next state, all actions)

State space:

The state space is preety big and we can think of tends to infinity.

State space tooks below parameters

* Source IP
* Destination IP
* Reward for load balancing request with persistent session hit.
* Reward for Load balancing last request.

Action Space:

In Fig 1. Any new incoming request can be sent to three forward proxy servers for load balancing. Therefore action space is equal to number of forward proxies provisioned.

Reward:

Count\_Array = array of integers representing number of live session on each proxy server.

persistent hit = bool variable representing if we were able to load balance a request to a server having a HTTP1.1 persistent session for same source IP and destination URL.

Rewards are calculated by considering below two parameters

* Reward for load balancing request with persistent session hit.
* Reward for Load balancing last request

Reward = variable to compute reward

Std\_dev = standard\_deviation\_of(Count\_Array)

Assert(Std\_dev >=0)

If Std\_dev <100:

Reward += 10

If (Std\_dev >100):

Reward – = 30

If persistent hit:

Reward += 30

else:

Reward -= 0

end

The request-response property Configuration:

The configuration required for a network simulation has too many environment variables. Therefore program implemented is written carefully such that tuning of environment does not require re-write of any code. This is achieved via .ini file format.

**CHAPTER 6**

**RESULTS**

Figure 6.1 shows Snap Shot of Generated IPs(s\_ip.txt)

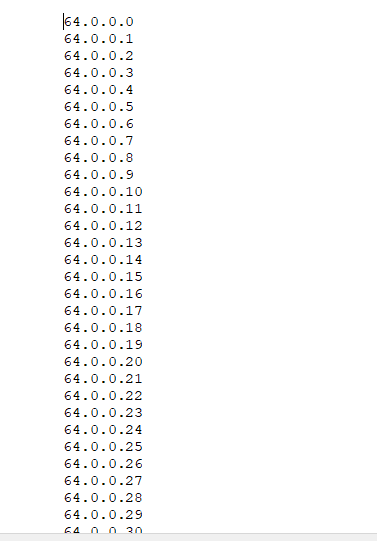


Fig 6.1 Snap shot of s\_ip.txt

Figure 6.2 shows Snap Shot of top 1million URLs(top-1m.csv)

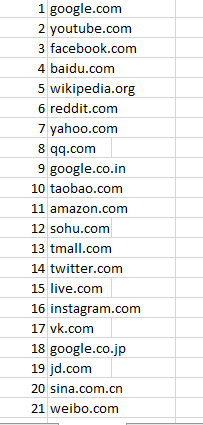


Fig 6.2 Snap Shot of top-1m.csv

Figure 6.3 Snap Shot of input Generation exp\_input.txt

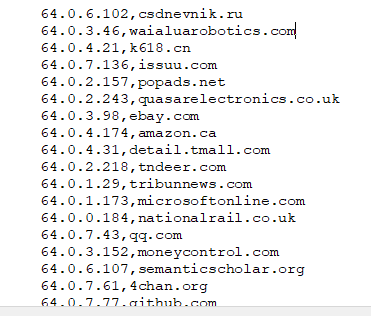


Fig 6.3 Snap Shot of exp\_input.txt

The effectiveness of Load balancing using Deep reinforcement learning policy and round robin policy has been compared based on,

Performance of Load balancer calculated from the standard deviation of number of requests send to each proxy server.in interval of 2000 sessions. the values are listed in fig 6.4

Session Persistency: To measure session persistency we have used Boolean variable, which shows if we were able to load balance a request to a server having a HTTP1.1 persistent session for same source IP and destination IP then it indicate the session is persistent otherwise its not persistent session.

The experiment is carried for 600000 sessions and Standard deviation has been calculated.

The above estimate for DRL and RR policy are depicted in below figures respectively



Fig 6.4 Result obtained by Deep Reinforcement Learning policy.

Same input is also applied to measure the performance of load balancer using Round robin Policy. Fig shows the performance of load balancer using round robin policy



Fig. 6.5 Snap shot of Results Obtained by Round Robin Policy

Fig 6.6. Performance of load balancer using DRL and Round Robin Policy.

As shown in figure 5.6 the performance of Deep Reinforcement learning policy and Round robin Policy are giving same results for initial 20000 sessions.

* Session Stickiness:

To measure session persistency we have used Boolean variable, which shows if we were able to load balance a request to a server having a HTTP1.1 persistent session for same source IP and destination IP then it indicate the session is persistent otherwise its not persistent session

This is specific to HTTP1.1 session persistence. It measures how many sessions has been persisted over total number of sessions. To check session Stickiness we have considered 600000 total sessions. Table 1 shows the result of session persistency using DRL and Round Robin Policy

Table 1. Session Stickiness using DRL & Round Robin policy

|  |  |  |
| --- | --- | --- |
| Total Session | DRL | Round Robin |
| 2000 | 588 | 610 |
| 10000 | 1671 | 1210 |
| 50000 | 1973 | 1972 |
| 100000 | 2000 | 2000 |
| 150000 | 2000 | 2000 |
| 200000 | 2000 | 2000 |
| 250000 | 2000 | 2000 |
| 300000 | 1999 | 2000 |
| 350000 | 1999 | 2000 |
| 400000 | 2000 | 2000 |
| 450000 | 1996 | 2000 |
| 500000 | 1998 | 2000 |
| 550000 | 1999 | 2000 |
| 600000 | 1998 | 1999 |

Fig 6.7. Session Stickiness by DRL & Round Robin Policy

**CHAPTER 7**

**CONCLUSION AND FUTURE ENHANCEMENTS**

* **Conclusion**

This project describes how Deep reinforcement learning algorithm can be used for network traffic management. It is seen that the Deep reinforcement algorithm achieves load balancing with performance similar to other existing load balancing policies, but without requiring human intervention. Another added advantage from the algorithm is that it achieves session persistence which may not be there in some of the existing policies.

* **Future Work**

We have setup and experimented with Deep Q Network(DQN) Learning based load balancing policy with server load balance factor acting as reinforcement reward. Results shows that such policy is easy to produce, simple, efficient and automatically adaptive to traffic pattern and network characteristics. We demonstrate that such policy can be deployed for real traffic handling. It is superior to hand crafted policies due to its agile nature and equivalent performance. The algorithm developed can work for forward HTTP proxies as well as reverse HTTP proxies. In fact, core algorithm is application independent and can be used for general load balance problem in variety of domains.

**CHAPTER 8**

**REFRENCES**

1. https://en.wikipedia.org/wiki/Load\_balancing\_(computing)
2. Pradeep Singh Rawat, G. P. Saroha, Yarun Barthwal ” Performance Evaluation of Social Networking Application with different Load balancing policy across Virtual Machine in a Single Data Center using Cloud Analyst” Parallel Distributed and Grid Computing (PDGC), 2012 2nd IEEE International Conference on
3. V Mnih, K Kavukcuoglu, D Silver, A Rusu, J Veness, M G Bellemare, A Graves, M Riedmiller, A Fidjeland, G Ostrovski, S Petersen, C Beattie, A Sadik, I Antonoglou, H King, D Kumaran, D Wierstra, S Legg, D Hassabis.”Human-level control through deep reinforcement learning” **Nature**, Vol. 518, No. 7540, 2015, pp. 529–533.
4. https://www.ibm.com/developerworks/library/ac-adapt/index.html
5. <https://www.digitalocean.com/community/tutorials/what-is-load-balancing>
6. Samreen, Faiza, and M. Sikandar Hayat Khiyal. "Q-Learning Scheduler and Load Balancer for Heterogeneous Systems." Journal of Applied Sciences 7, no. 11 (2007): 1504-1510.
7. Kim, Hye-Young, and Jong-Min Kim. "A load balancing scheme based on deep-learning in IoT." Cluster Computing20, no. 1 (2017): 873-878.
8. Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, “Playing Atari with Deep Reinforcement Learning” NIPS Jan. 1, 2013
9. Mnih, Volodymyr, Badia, Adri Puigdomnech, Mirza, Mehdi, Graves, Alex, Lillicrap, Timothy P., Harley, Tim, Silver, David, and Kavukcuoglu, Koray. “Asynchronous methods for deep reinforcement learning”. **ICML**, 2016
10. Irina Higgins, Arka Pal, Andrei Rusu, Loic Matthey, Christopher Burgess, Alexander Pritzel, Matthew Botvinick, Charles Blundell, Alexander Lerchner, DARLA: Improving Zero-Shot Transfer in Reinforcement Learning, The 34th International Conference on Machine Learning (ICML 2017), Sydney, Australia, August 2017.
11. [Hye-Young Kim](https://dl.acm.org/author_page.cfm?id=81442609456&coll=DL&dl=ACM&trk=0), Jong-Min Kim”A load balancing scheme based on deep-learning in IoT” Cluster Computing, Volume 20 Issue 1, March 2017 Pages 873-878, **ACM** Digital library.
12. Nada M. Al Sallami , Ali Al Daoud , Sarmad A. Al Alousi “ Load Balancing with Neural Network” (IJACSA) International Journal of Advanced Computer Science and Applications, Vol. 4, No. 10, 2013
13. Deep Learning Blog: <https://keon.io/deep-q-learning/>