

TABLE OF CONTENTS

SL. NO	CHAPTER NAME	PAGE NO.
1	INTRODUCTION	5-6
	1.1 SCOPE OF WORK	5
	1.2 MOTIVATION	6
2	LITERATURE SURVEY	8-13
	2.1 REINFORCEMENT LEARNING	8
	2.2 Q LEARNING	9
	2.3 MARKOV DECISION PROCESS	9
	2.4 TENSORFLOW	10
3	SYSTEM REQUIREMENTS AND SPECIFICATION	15-18
	3.1 FUNTIONAL REQUIREMENTS	15
	3.2 NON FUNCTIONAL REQUIREMENTS	15
	3.3 OTHER NON FUNCTIONAL REQUIREMENTS	16
	3.4 DEVELOPMENT REQUIREMENTS	17
4	KEY FEATURES AND COMPONENTS	20-24
	4.1 REINFORCEMENT LEARNING	20
	4.2 OPENAI	22
	4.3 OPENAI GYM	23
	4.4 ATARI	23
	4.5 MULTI ARMED BANDIT	24
	4.6 MODEL BASED RL	24
	4.7 ACTOR CRITIC A3C	24
5	PROPOSED SYSTEM AND FRAMEWORK	26-31
	5.1 PROPOSED SYSTEM	26
	5.2 FRAMEWORK	30
6	SYSTEM ANALYSIS AND DESIGN	33-36
	6.1 SYSTEM ANALYSIS	33

	6.2 SYSTEM DESIGN	36
7	IMPLEMENTATION	38-41
	7.1 IMPLEMENTATION	38
	7.2 EXECUTION PHASE	40
8	EXECUTION AND TEST CASES	43-45
	8.1 EXECUTION	43
9	CONCLUSION AND FUTURE ENHANCEMENTS	47-48
	9.1 CONCLUSION	47
	9.2 FUTURE WORK	47
	9.3 PROJECT ACTIVITY	48
10	REFERENCES	50

TABLE OF FIGURES

SL. NO	DESCRIPTION	PAGE NO
1	INTERNALL MOTIVATED BEHAVIOR	27
2	DQN VS A3C	28
3	PARALLEL DQN	29
4	OVERVIEW OF THE FRAMEWORK	30
5	SDLC	33
6	LOW LEVEL SYSTEM DESIGN	36
7	HIGH LEVEL DESIGN	36
8	SNAP OF GUI	43
9	SNAP OF SPACE INVADER	44
10	SNAP OF PONG GAME	44
11	QUALITY GRAHS	45
12	RUNTIME GRAPH	45
13	GANTT CHART	48

CHAPTER 1

INTRODUCTION

CHAPTER 1

INTRODUCTION

1. INTRODUCTION

The method of learning goal-directed behaviour in environments with sparse feedback is a major challenge for reinforcement learning algorithms. The primary difficulty arises due to insufficient exploration, resulting in an agent being unable to learn robust value functions. Intrinsically motivated agents can explore new behaviour for its own sake rather than to directly solve problems. Such intrinsic behaviours could eventually help the agent solve tasks posed by the environment. We present a parallel-DQN, a framework operating at different temporal scales, with intrinsically motivated deep reinforcement learning. A top-level value function learns a policy over intrinsic goals, and a lower-level function learns a policy over atomic actions to satisfy the given goals. This provides an efficient space for exploration in complicated environments.

1.1 SCOPE OF WORK

Efficient exploration is fundamentally about managing uncertainties. In reinforcement learning, these uncertainties pertain to the unknown nature of the reward and transition functions. The traditional unit of frequentist certainty is undoubtedly the count: an integer describing the number of observations of a certain type. Most Bayesian exploration schemes use counts in their posterior estimates, for example when the uncertainty over transition probabilities is captured by an exponential family prior. In simple domains, efficient exploration is theoretically well-understood, if not yet solved. There are near-optimal algorithms for multi-armed bandits; the hardness of exploration in Markov Decision Processes (MDPs) is well-understood. By stark contrast, contemporary practical successes in reinforcement learning still rely on simple forms of exploration, for example -greedy policies – what Thrun (1992) calls undirected exploration.

1.2 MOTIVATION

Reinforcement learning is a type of Machine Learning algorithms which allows software agents and machines to automatically determine the ideal behaviour within a specific context, to maximize its performance.

The motivation was to develop a framework with organized deep reinforcement learning modules working at an environment agnostic level. Teaching AI to play games can be used to measure how smart an AI actually is! Earlier board games could be taught to beat using pure brute force crunching large number of moves, but games like Go can't be solved that way, they are learnt by the machines to play via heuristics. Alpha Go uses deep neural network (looks for patterns) Alpha GO has 48 layers of neurons. (it mimics human intuition and creativity). But a program like Alpha GO is a narrow AI which is focused on tackling and beating the sharpest human mind at the hardest board game. Our attempt is to try and achieve a foothold in trying to achieve this with games which are much more complex to realise than board games and to try and enforce Reinforcement Learning to solve problem skills and learn to play the game, running on Atari environments, trying to take the Machine Learning applications one step further.

CHAPTER 2

LITERATURE SURVEY

CHAPTER 2

LITERATURE SURVEY

Deep Reinforcement Learning has recently become a really hot area of research, due to the huge amount of breakthroughs in last couple of years. By feeding sufficient data into deep neural networks, it is often possible to learn better representations than handcrafted features. Our goal is to connect a reinforcement learning algorithm to a deep neural network which operates directly on RGB images and efficiently process training data by using stochastic gradient updates. The main idea of DQN is to compress Q-table by learning to recognize in-game objects and their behaviour, in order to predict delayed reward for each action given the state.

2.1 REINFORCEMENT LEARNING

Reinforcement learning (RL) is an area of machine learning inspired by behaviorist psychology, concerned with how software agents ought to take actions in an environment so as to maximize some notion of cumulative reward. The problem, due to its generality, is studied in many other disciplines, such as game theory, control theory, operations research, information theory, simulation-based optimization, multi-agent systems, swarm intelligence, statistics, and genetic algorithms. In the operations research and control literature, the field where reinforcement learning methods are studied is called approximate dynamic programming. The problem has been studied in the theory of optimal control, though most studies are concerned with the existence of optimal solutions and their characterization, and not with the learning or approximation aspects. In economics and game theory, reinforcement learning may be used to explain how equilibrium may arise under bounded rationality.

In machine learning, the environment is typically formulated as a Markov decision process (MDP), as many reinforcement learning algorithms for this context utilize dynamic programming techniques. The main difference between the classical techniques and reinforcement learning algorithms is that the latter do not need knowledge about the MDP and they target large MDPs where exact methods become infeasible.

Reinforcement learning is a type of Machine Learning algorithms which allows software agents and machines to automatically determine the ideal behaviour within a specific context, to maximize its performance.

Reinforcement algorithms are not given explicit goals; instead, they are forced to learn these optimal goals by trial and error. Think of the classic Mario Bros. video game; reinforcement learning algorithms would, by trial and error, determine that certain movements and button pushes would advance the player's standing in the game, and trial and error would aim to result in an optimal state of game play.

2.2 Q LEARNING

Q-learning is a model-free reinforcement learning technique. Specifically, Q-learning can be used to find an optimal action-selection policy for any given (finite) Markov decision process (MDP). It works by learning an action-value function that ultimately gives the expected utility of taking a given action in a given state and following the optimal policy thereafter. A policy is a rule that the agent follows in selecting actions, given the state it is in. When such an action-value function is learned, the optimal policy can be constructed by simply selecting the action with the highest value in each state. One of the strengths of Q-learning is that it is able to compare the expected utility of the available actions without requiring a model of the environment. Additionally, Q-learning can handle problems with stochastic transitions and rewards, without requiring any adaptations. It has been proven that for any finite MDP, Q-learning eventually finds an optimal policy, in the sense that the expected value of the total reward return over all successive steps, starting from the current state, is the maximum achievable.

2.3 MARKOV DECISION PROCESS

Markov decision processes (MDPs) provide a mathematical framework for modeling decision making in situations where outcomes are partly random and partly under the control of a decision maker. MDPs are useful for studying a wide range of optimization problems solved via dynamic programming and reinforcement learning. MDPs were known at least as early as the 1950s; a core body of research on Markov decision processes resulted from Ronald A. Howard's book published in 1960, *Dynamic Programming and Markov Processes*. They are used in a wide area of disciplines, including robotics, automated

control, economics, and manufacturing more precisely, a Markov Decision Process is a discrete time stochastic control process.

The core problem of MDPs is to find a "policy" for the decision maker: a function f that specifies the action $f(s)$ that the decision maker will choose when in state s . Note that once a Markov decision process is combined with a policy in this way, this fixes the action for each state and the resulting combination behaves like a Markov chain.

2.4 TENSORFLOW

TensorFlow is an open source software library for machine learning across a range of tasks, and developed by Google to meet their needs for systems capable of building and training neural networks to detect and decipher patterns and correlations, analogous to the learning and reasoning which humans use. It is currently used for both research and production at Google products, often replacing the role of its closed-source predecessor, DistBelief. TensorFlow was originally developed by the Google Brain team for internal Google use before being released under the Apache 2.0 open source license on November 9, 2015.

TensorFlow is Google Brain's second generation machine learning system, released as open source software on November 9, 2015. While the reference implementation runs on single devices, TensorFlow can run on multiple CPUs and GPUs (with optional CUDA extensions for general-purpose computing on graphics processing units). TensorFlow is available on 64-bit Linux, macOS, Windows, and mobile computing platforms including Android and iOS.

TensorFlow computations are expressed as stateful dataflow graphs. The name TensorFlow derives from the operations which such neural networks perform on multidimensional data arrays. These multidimensional arrays are referred to as "tensors". In June 2016, Google's Jeff Dean stated that 1,500 repositories on GitHub mentioned TensorFlow, of which only 5 were from Google.

TensorFlow provides a Python API, as well as C++, Haskell, Java, Go, and Rust APIs. In addition, there is a 3rd party package for R.

Among the applications for which TensorFlow is the foundation, are automated image captioning software, such as DeepDream. Google officially implemented RankBrain on 26

October 2015, backed by TensorFlow. RankBrain now handles a substantial number of search queries, replacing and supplementing traditional static algorithm based search results.

2.5 OPENAI

OpenAI is a non-profit artificial intelligence (AI) research company, associated with business magnate Elon Musk, that aims to carefully promote and develop friendly AI in such a way as to benefit humanity as a whole. The organization aims to "freely collaborate" with other institutions and researchers by making its patents and research open to the public. The company is supported by over US\$1 billion in commitments; however, only a tiny fraction of the \$1 billion pledged is expected to be spent in the first few years. The founders are motivated in part by concerns about existential risk from artificial general intelligence.

OpenAI's founders structured it as a non-profit free of financial stockholder obligations, so that they could focus its research on creating a positive long-term human impact.

OpenAI states that "it's hard to fathom how much human-level AI could benefit society," and that it's equally difficult to comprehend "how much it could damage society if built or used incorrectly". Research on safety cannot safely be postponed: "because of AI's surprising history, it's hard to predict when human-level AI might come within reach." OpenAI states that AI "should be an extension of individual human wills and, in the spirit of liberty, as broadly and evenly distributed as possible..." Co-chair Sam Altman expects the decades-long project to surpass human intelligence.

2.6 OPENAI GYM

On April 27, 2016, OpenAI released a public beta of "OpenAI Gym", a platform for reinforcement learning research that aims to provide an easy-to-setup general-intelligence benchmark with a wide variety of different environments (somewhat akin to, but broader than, the ImageNet Large Scale Visual Recognition Challenge used in supervised learning research), and that hopes to standardize the way in which environments are defined in AI research publications, so that published research becomes more easily reproducible. The project claims to provide the user with a simple interface. As of June 2017, "OpenAI Gym" can only be used through Python, but more languages are coming soon.

2.7 ATARI

The original Atari, Inc. founded in 1972 by Nolan Bushnell and Ted Dabney was a pioneer in arcade games, home video game consoles, and home computers. The company's products, such as *Pong* and the Atari 2600, helped define the electronic entertainment industry from the 1970s to the mid-1980s.

In 1984, the original Atari Inc. was split due to its role in the video game crash of 1983, and the arcade division was turned into Atari Games Inc. Atari Games received the rights to use the logo and brand name with appended text "Games" on arcade games, as well as rights to the original 1972–1984 arcade hardware properties. The Atari Consumer Electronics Division properties were in turn sold to Jack Tramiel's Tramel Technology Ltd., which then renamed itself to Atari Corporation. In 1996, Atari Corporation reverse-merged with disk-drive manufacturer JT Storage (JTS), becoming a division within the company.

Atari, Inc. was an American video game developer and home computer company founded in 1972 by Nolan Bushnell and Ted Dabney. Primarily responsible for the formation of the video arcade and modern video game industries, the company was closed and its assets split in 1984 as a direct result of the North American video game crash of 1983.

2.8 MULTI-ARMED BANDIT

In probability theory, the multi-armed bandit problem (sometimes called the K- or N-armed bandit problem) is a problem in which a gambler at a row of slot machines (sometimes known as "one-armed bandits") has to decide which machines to play, how many times to play each machine and in which order to play them. When played, each machine provides a random reward from a probability distribution specific to that machine. The objective of the gambler is to maximize the sum of rewards earned through a sequence of lever pulls.

2.9 MODEL BASED RL

Unlike in computer simulations, physical environments take time to navigate, and the physical rules of the world prevent things like easy environment resets from being feasible. Instead, we can save time and energy by building a model of the environment. With such a model, an agent can ‘imagine’ what it might be like to move around the real environment, and we can train a policy on this imagined environment in addition to the real one. If we were

given a good enough model of an environment, an agent could be trained entirely on that model, and even perform well when placed into a real environment for the first time.

2.10 ASYNCHRONOUS ACTOR CRITIC AGENTS (A3C)

A3C is conceptually simple and lightweight framework for deep reinforcement learning that uses asynchronous gradient descent for optimization of deep neural network controllers. Parallel actor-learners have a stabilizing effect on training allowing all four methods to successfully train neural network controllers. The best performing method, an asynchronous variant of actor-critic, surpasses the current state-of-the-art on the Atari domain while training for half the time on a single multi-core CPU instead of a GPU.

CHAPTER 3

SYSTEM REQUIREMENTS

AND SPECIFICATIONS

CHAPTER 3

SYSTEM REQUIREMENTS SPECIFICATION

The software requirements specification document enlists all necessary requirements that are required for the project development. To derive the requirements we need to have clear and thorough understanding of the products to be developed. This is prepared after detailed communications with the project team and customer.

3.1 FUNCTIONAL REQUIREMENTS

The Functional Requirements Specification documents the operations and activities that a system must be able to perform.

- 1) Take an environment as input
- 2) Learn the game
- 3) Optimise Q network

3.2 NON-FUNCTIONAL REQUIREMENTS

The modeling, automatic implementation and runtime verification of constraints in component-based applications. Constraints have been assuming an ever more relevant role in modeling distributed systems as long as business rules implementation, design-by-contract practice, and fault-tolerance requirements are concerned. Nevertheless, component developers are not sufficiently supported by existing tools to model and implement such features, we propose a methodology and a set of tools that enable developers both to model component constraints and to generate automatically component skeletons that already implement such constraints. The methodology has been extended to support implementation even in case of legacy components.

3.3 OTHER NON-FUNCTIONAL REQUIREMENTS

3.3.1 Safety and Security Requirements

Many software-intensive systems have significant safety and security ramifications and need to have their associated safety- and security-related requirements properly engineered. For example, it has been observed by several consultants, researchers, and authors that inadequate requirements are a major cause of accidents involving software-intensive systems. Yet in practice, there is very little interaction between the requirements, safety and security disciplines and little collaboration between their respective communities. Most requirements engineers know little about safety and security engineering, and most safety and security engineers know little about requirements engineering. Also, safety and security engineering typically concentrates on architectures and designs rather than requirements because hazard and threat analysis typically depend on the identification of vulnerable hardware and software components, the exploitation of which can cause accidents and enable successful attacks.

3.3.2 Software Quality Attributes

Following factors are used to measure software development quality. Each attribute can be used to measure the product performance. These attributes can be used for Quality assurance as well as Quality control. Quality Assurance activities are oriented towards prevention of introduction of defects and Quality control activities are aimed at detecting defects in products and services.

Reliability

Measure if product is reliable enough to sustain in any condition give consistently correct results. Product reliability is measured in terms of working of project under different working environment and different conditions.

Maintainability

Different versions of the product should be easy to maintain. For development it should be easy to add code to existing system, should be easy to upgrade for new features and

new technologies time to time. Maintenance should be cost effective and easy. System be easy to maintain and correcting defects or making a change in the software

Usability

This can be measured in terms of ease of use. Application should be user friendly. The system must be Easy to use for input preparation, operation, and interpretation of output, and provide consistent user interface standards or conventions with our other frequently used systems. They should be easy for new or infrequent users to learn to use the system.

Portability

This can be measured in terms of Costing issues related to porting, Technical issues related to porting, Behavioral issues related to porting

Correctness

Application should be correct in terms of its functionality, calculations used internally and the navigation should be correct. This means application should adhere to functional requirements.

Efficiency

To Major system quality attribute. Measured in terms of time required to complete any task given to the system. For example system should utilize processor capacity, disk space and memory efficiently. If system is using all the available resources then user will get degraded performance failing the system for efficiency. If system is not efficient then it cannot be used in real time applications.

3.4 DEVELOPMENT REQUIREMENTS

Hardware Requirement

PROCESSOR : Intel i5

RAM : 4GB

MONITOR : 15"

HARD DISK : 80 GB

KEYBOARD : STANDARD 102 KEYS

Software Requirement

OPERATING SYSTEM : LINUX

PROGRAMMING LANGUAGE : PYTHON

IDE : SUBLIME TEXT

CHAPTER 4

KEY FEATURES AND COMPONENTS

CHAPTER 4

KEY FEATURES AND COMPONENTS

4.1 REINFORCEMENT LEARNING

Reinforcement learning (RL) is an area of machine learning inspired by behaviorist psychology, concerned with how software agents ought to take actions in an environment so as to maximize some notion of cumulative reward. The problem, due to its generality, is studied in many other disciplines, such as game theory, control theory, operations research, information theory, simulation-based optimization, multi-agent systems, swarm intelligence, statistics, and genetic algorithms. In the operations research and control literature, the field where reinforcement learning methods are studied is called approximate dynamic programming. The problem has been studied in the theory of optimal control, though most studies are concerned with the existence of optimal solutions and their characterization, and not with the learning or approximation aspects. In economics and game theory, reinforcement learning may be used to explain how equilibrium may arise under bounded rationality.

In machine learning, the environment is typically formulated as a Markov decision process (MDP), as many reinforcement learning algorithms for this context utilize dynamic programming techniques. The main difference between the classical techniques and reinforcement learning algorithms is that the latter do not need knowledge about the MDP and they target large MDPs where exact methods become infeasible.

Reinforcement learning is a type of Machine Learning algorithms which allows software agents and machines to automatically determine the ideal behaviour within a specific context, to maximize its performance.

Reinforcement algorithms are not given explicit goals; instead, they are forced to learn these optimal goals by trial and error. Think of the classic Mario Bros. video game; reinforcement learning algorithms would, by trial and error, determine that certain movements and button pushes would advance the player's standing in the game, and trial and error would aim to result in an optimal state of game play.

4.2 Q LEARNING

Q-learning is a model-free reinforcement learning technique. Specifically, Q-learning can be used to find an optimal action-selection policy for any given (finite) Markov decision process (MDP). It works by learning an action-value function that ultimately gives the expected utility of taking a given action in a given state and following the optimal policy thereafter. A policy is a rule that the agent follows in selecting actions, given the state it is in. When such an action-value function is learned, the optimal policy can be constructed by simply selecting the action with the highest value in each state. One of the strengths of Q-learning is that it is able to compare the expected utility of the available actions without requiring a model of the environment. Additionally, Q-learning can handle problems with stochastic transitions and rewards, without requiring any adaptations. It has been proven that for any finite MDP, Q-learning eventually finds an optimal policy, in the sense that the expected value of the total reward return over all successive steps, starting from the current state, is the maximum achievable.

4.3 MARKOV DECISION PROCESS

Markov decision processes (MDPs) provide a mathematical framework for modeling decision making in situations where outcomes are partly random and partly under the control of a decision maker. MDPs are useful for studying a wide range of optimization problems solved via dynamic programming and reinforcement learning. MDPs were known at least as early as the 1950s; a core body of research on Markov decision processes resulted from Ronald A. Howard's book published in 1960, *Dynamic Programming and Markov Processes*. They are used in a wide area of disciplines, including robotics, automated control, economics, and manufacturing more precisely, a Markov Decision Process is a discrete time stochastic control process.

The core problem of MDPs is to find a "policy" for the decision maker: a function f that specifies the action $f(s)$ that the decision maker will choose when in state s . Note that once a Markov decision process is combined with a policy in this way, this fixes the action for each state and the resulting combination behaves like a Markov chain.

4.4 TENSORFLOW

TensorFlow is an open source software library for machine learning across a range of tasks, and developed by Google to meet their needs for systems capable of building and training neural networks to detect and decipher patterns and correlations, analogous to the learning and reasoning which humans use. It is currently used for both research and production at Google products, often replacing the role of its closed-source predecessor, DistBelief. TensorFlow was originally developed by the Google Brain team for internal Google use before being released under the Apache 2.0 open source license on November 9, 2015.

TensorFlow is Google Brain's second generation machine learning system, released as open source software on November 9, 2015. While the reference implementation runs on single devices, TensorFlow can run on multiple CPUs and GPUs (with optional CUDA extensions for general-purpose computing on graphics processing units). TensorFlow is available on 64-bit Linux, macOS, Windows, and mobile computing platforms including Android and iOS.

TensorFlow computations are expressed as stateful dataflow graphs. The name TensorFlow derives from the operations which such neural networks perform on multidimensional data arrays. These multidimensional arrays are referred to as "tensors". In June 2016, Google's Jeff Dean stated that 1,500 repositories on GitHub mentioned TensorFlow, of which only 5 were from Google.

Among the applications for which TensorFlow is the foundation, are automated image captioning software, such as DeepDream. Google officially implemented RankBrain on 26 October 2015, backed by TensorFlow. RankBrain now handles a substantial number of search queries, replacing and supplementing traditional static algorithm based search results.

4.5 OPENAI

OpenAI is a non-profit artificial intelligence (AI) Research Company, associated with business magnate Elon Musk that aims to carefully promote and develop friendly AI in such a way as to benefit humanity as a whole. The organization aims to "freely collaborate" with other institutions and researchers by making its patents and research open to the public. The company is supported by over US\$1 billion in commitments; however, only a tiny fraction of the \$1 billion pledged is expected to be spent in the first few years. The founders are motivated in part by concerns about existential risk from artificial general intelligence.

OpenAI's founders structured it as a non-profit free of financial stockholder obligations, so that they could focus its research on creating a positive long-term human impact.

OpenAI states that "it's hard to fathom how much human-level AI could benefit society," and that it's equally difficult to comprehend "how much it could damage society if built or used incorrectly". Research on safety cannot safely be postponed: "because of AI's surprising history, it's hard to predict when human-level AI might come within reach." OpenAI states that AI "should be an extension of individual human wills and, in the spirit of liberty, as broadly and evenly distributed as possible..." Co-chair Sam Altman expects the decades-long project to surpass human intelligence.

4.6 OPENAI GYM

On April 27, 2016, OpenAI released a public beta of "OpenAI Gym", a platform for reinforcement learning research that aims to provide an easy-to-setup general-intelligence benchmark with a wide variety of different environments (somewhat akin to, but broader than, the ImageNet Large Scale Visual Recognition Challenge used in supervised learning research), and that hopes to standardize the way in which environments are defined in AI research publications, so that published research becomes more easily reproducible. The project claims to provide the user with a simple interface. As of June 2017, "OpenAI Gym" can only be used through Python, but more languages are coming soon.

4.7 ATARI

The original Atari, Inc. founded in 1972 by Nolan Bushnell and Ted Dabney was a pioneer in arcade games, home video game consoles, and home computers. The company's products, such as *Pong* and the Atari 2600, helped define the electronic entertainment industry from the 1970s to the mid-1980s.

In 1984, the original Atari Inc. was split due to its role in the video game crash of 1983, and the arcade division was turned into Atari Games Inc. Atari Games received the rights to use the logo and brand name with appended text "Games" on arcade games, as well as rights to the original 1972–1984 arcade hardware properties. The Atari Consumer Electronics Division properties were in turn sold to Jack Tramiel's Tramiel Technology Ltd., which then renamed itself to Atari Corporation. In 1996, Atari Corporation reverse-merged with disk-drive manufacturer JT Storage (JTS), becoming a division within the company.

Atari, Inc. was an American video game developer and home computer company founded in 1972 by Nolan Bushnell and Ted Dabney. Primarily responsible for the formation of the video arcade and modern video game industries, the company was closed and its assets split in 1984 as a direct result of the North American video game crash of 1983.

4.8 MULTI-ARMED BANDIT

In probability theory, the multi-armed bandit problem (sometimes called the K- or N-armed bandit problem) is a problem in which a gambler at a row of slot machines (sometimes known as "one-armed bandits") has to decide which machines to play, how many times to play each machine and in which order to play them. When played, each machine provides a random reward from a probability distribution specific to that machine. The objective of the gambler is to maximize the sum of rewards earned through a sequence of lever pulls.

4.9 MODEL BASED RL

Unlike in computer simulations, physical environments take time to navigate, and the physical rules of the world prevent things like easy environment resets from being feasible. Instead, we can save time and energy by building a model of the environment. With such a model, an agent can 'imagine' what it might be like to move around the real environment, and we can train a policy on this imagined environment in addition to the real one. If we were given a good enough model of an environment, an agent could be trained entirely on that model, and even perform well when placed into a real environment for the first time.

4.10 ASYNCHRONOUS ACTOR CRITIC AGENTS (A3C)

A3C is conceptually simple and lightweight framework for deep reinforcement learning that uses asynchronous gradient descent for optimization of deep neural network controllers. Parallel actor-learners have a stabilizing effect on training allowing all four methods to successfully train neural network controllers. The best performing method, an asynchronous variant of actor-critic, surpasses the current state-of-the-art on the Atari domain while training for half the time on a single multi-core CPU instead of a GPU.

CHAPTER 5

PROPOSED SYSTEM & FRAMEWORK

CHAPTER 5

PROPOSED SYSTEM AND FRAMEWORK

5.1 PROPOSED SYSTEM

We define the extrinsic reward function. The objective of the agent is to maximize this function over long periods of time. For example, this function can take the form of the agent's survival time or score in a game.

Agents

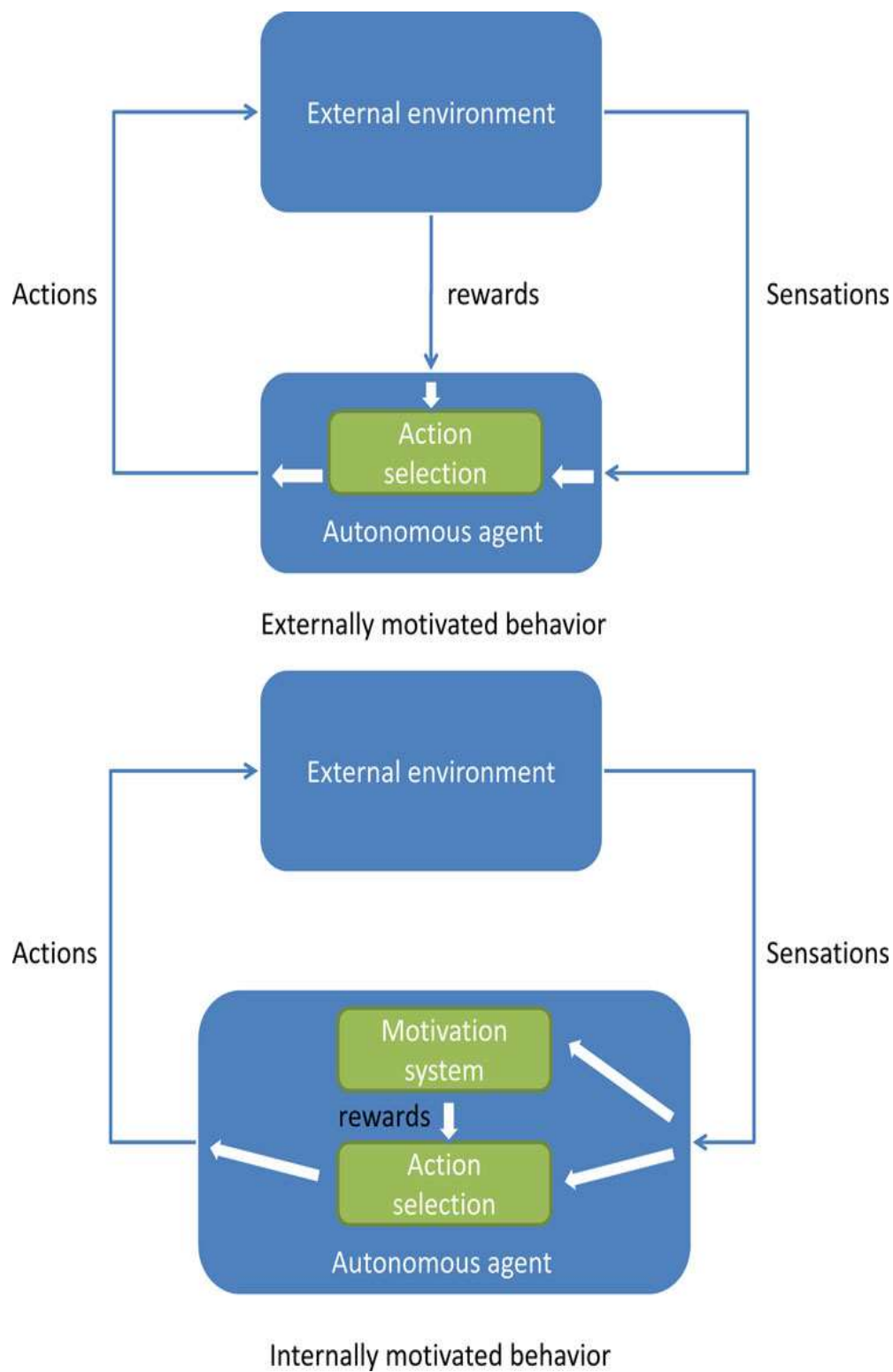
Effective exploration in MDPs is a significant challenge in learning good control policies. Methods such as greedy are useful for local exploration but fail to provide impetus for the agent to explore different areas of the state space. In order to tackle this, we utilize a notion of goals, which provide intrinsic motivation for the agent.

Temporal Abstractions

The agent uses a two-stage hierarchy consisting of a controller and a meta-controller. The meta-controller receives state s and chooses a goal $g \in G$, where G denotes the set of all possible current goals. The controller then selects an action a using s and g . The goal g remains in place for the next few time steps either until it is achieved or a terminal state is reached. The internal critic is responsible for evaluating whether a goal has been reached and providing an appropriate reward $r(g)$ to the controller.

Intrinsically Motivated Agents

These agents can explore new behaviour for its own sake rather than to directly solve problems. Such intrinsic behaviours could eventually help the agent solve tasks posed by the environment.

**Fig 1: Internally Motivated Behavior**

Value Function

A top-level value function learns a policy over intrinsic goals, and a lower-level function learns a policy over atomic actions to satisfy the given goals.

Parallel-DQN

Parallel-DQN allows for the trade-off between exploration vs exploitation by allowing parallel agents to modify the global network SIMULTANEOUSLY.

This provides an efficient space for exploration in complicated environments.

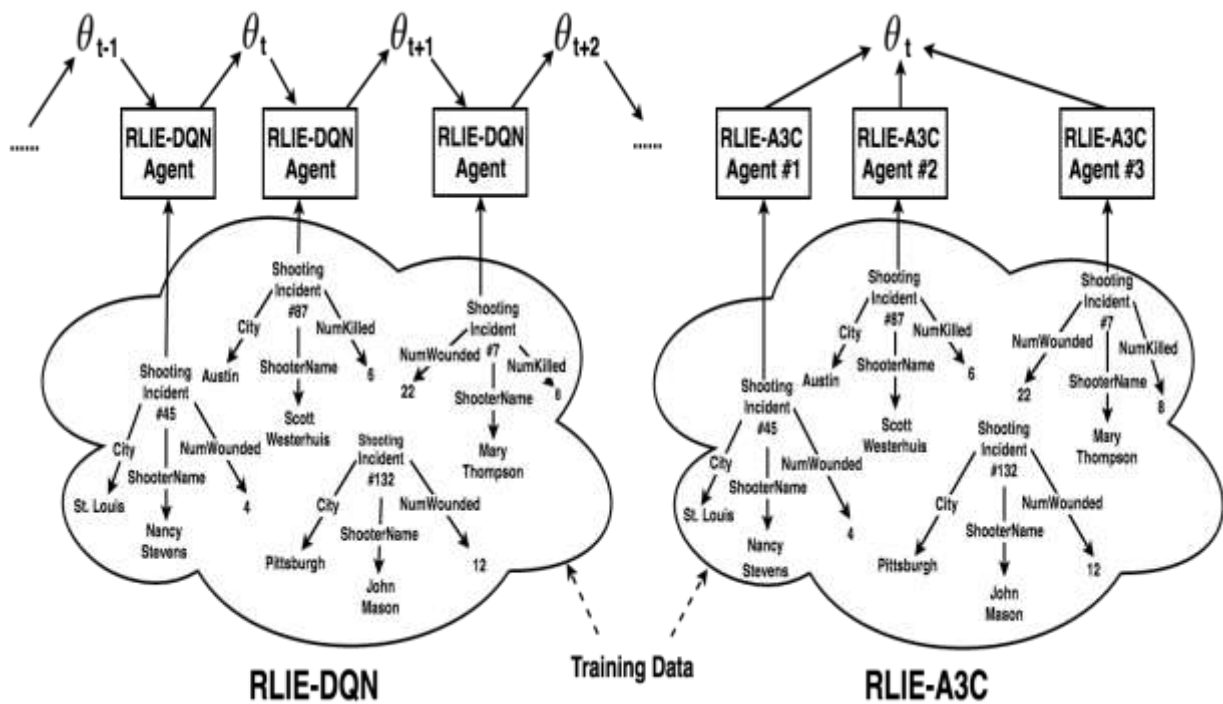


Fig 2: DQN vs Parallel DQN (A3C)

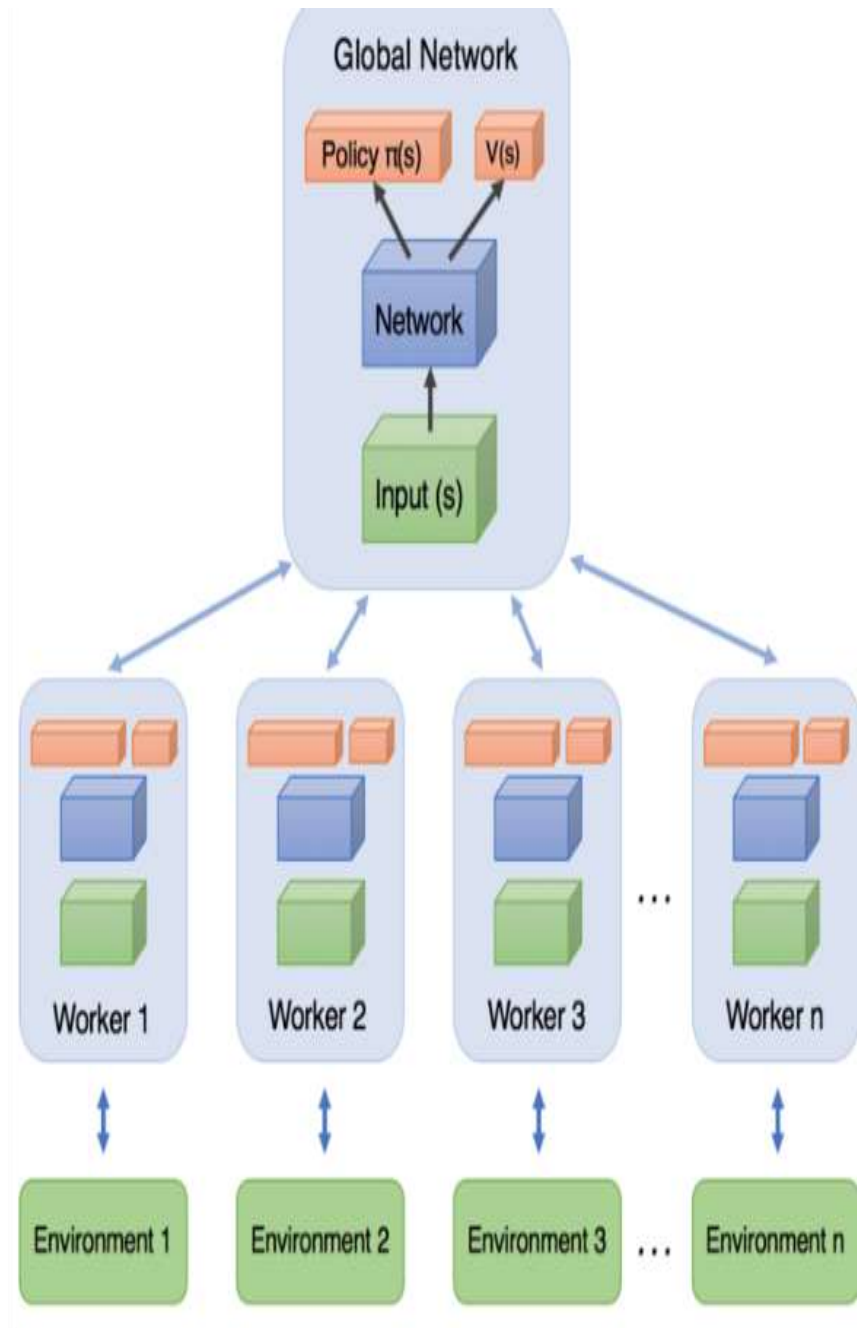


Fig 3: Parallel DQN Learning Policy Gradient as well as Value Functions

5.2 FRAMEWORK

We use the Deep Q-Learning framework to learn policies for both the controller and the meta-controller.

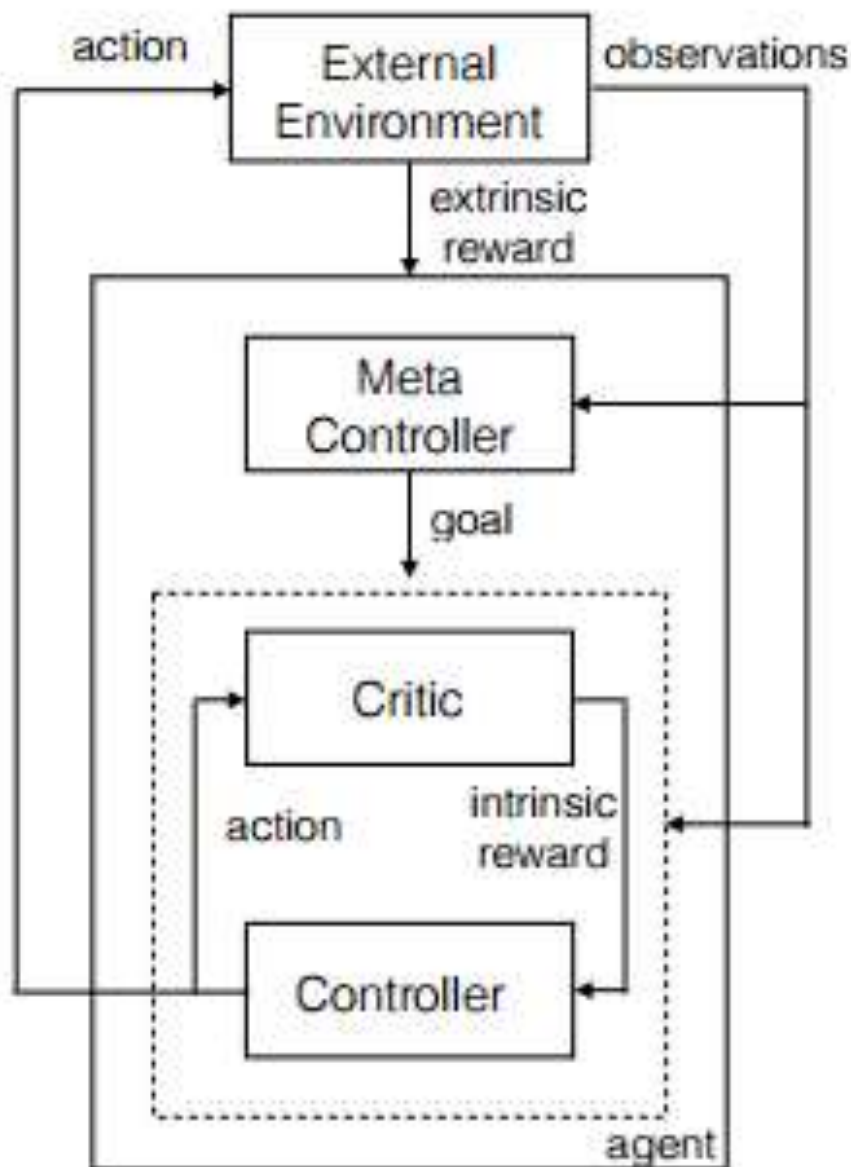


Fig 4: Overview of the framework

The agent produces actions and receives sensory observations. Separate deep-Q networks are used inside the meta-controller and controller. The meta-controller that looks at the raw states and produces a policy over goals by estimating the value function (by maximizing expected

future extrinsic reward). The controller takes in states and the current goal, and produces a policy over actions by estimating the value function to solve the predicted goal (by maximizing expected future intrinsic reward). The internal critic checks if goal is reached and provides an appropriate intrinsic reward to the controller. The controller terminates either when the episode ends or when g is accomplished. The meta-controller then chooses a new g and the process repeats.

CHAPTER 6

SYSTEM ANALYSIS & DESIGN

CHAPTER 6

SYSTEM ANALYSIS AND DESIGN

6.1 SYSTEM ANALYSIS

6.1.1 System Development Life Cycle

The SDLC is an application systems approach to development of information system. The tools of SDLC are using diagrams so it will be easier to understand, its stages related to each other. When changes occur in all phases of the system then it does not repeat again, SDLC phase is simpler.

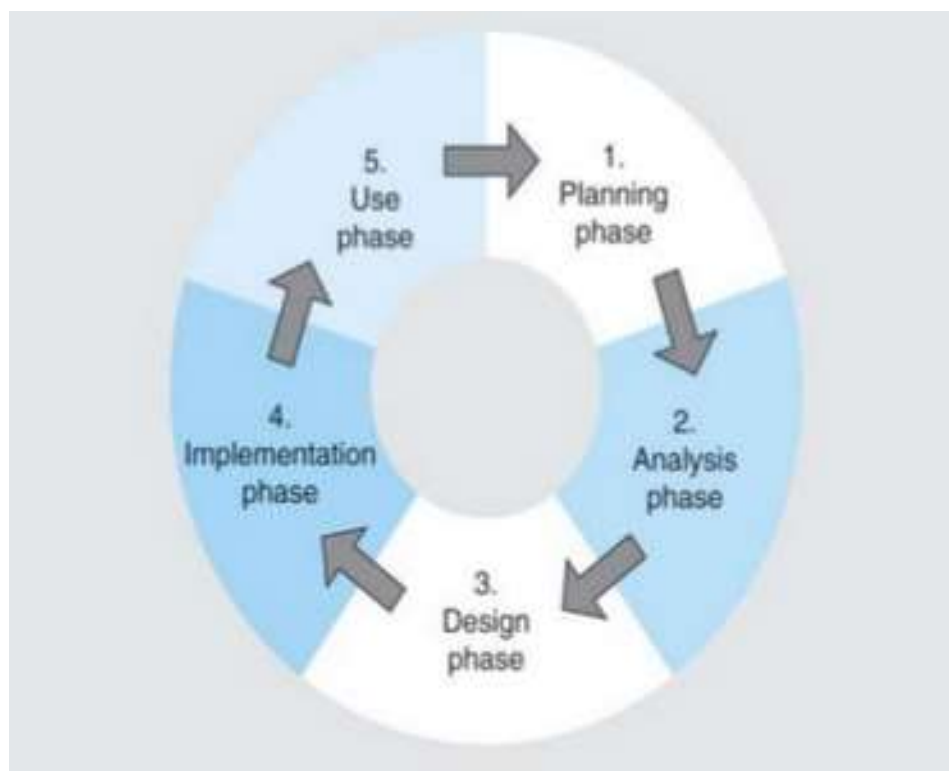


Fig 5: Software Development Life Cycle

6.1.2 The Development Phases in Brief

a.) Planning Phase

The Planning phase began with figuring out a way to develop a framework with organized deep reinforcement learning modules working at an environment agnostic level. And thus, to implement this model we planned to build an environment which will emulate and learn to play Games which run in the Atari Environment, and with our observations and models acquired by training the model we will be able to implement the concept of Reinforcement Learning by the means of Markov's Decision Process into our systems and learn to play the game perfectly.

b.) Analysis Phase

This phase began with the analysis of the requirements needed to build the working model of our autonomous game playing system, by preparing an abstract and carrying out a literature survey and realizing about the different approaches to go about and develop the system. The analysis revealed that the primary difficulty arises due to insufficient exploration, resulting in an agent being unable to learn robust value functions. Intrinsically motivated agents can explore new behavior for its own sake rather than to directly solve problems. Such intrinsic behaviors could eventually help the agent solve tasks posed by the environment. We present hierarchical-DQN (h-DQN), a framework to integrate hierarchical value functions, operating at different temporal scales, with intrinsically motivated deep reinforcement learning.

c.) Design Phase

We constructed our design by partitioning the design of the whole system into two levels, the high level design and the low level design. In the High level we presented a system represented in its most abstract level which consisted of the methodologies and necessary approaches to go about designing the environment, agents and the convolutional neural networks. In the lower layer of the design we have focus mainly on the build and the detailed

design of the Convolutional Neural Network and how it could perhaps be used to train the model using the tensors.

d.) Implementation Phase

The Implementation phase in order to develop a working model was divided into three segments, namely, common, simulator, & train-Atari. In the first segment we handle parameters which are common throughout all the games running in the Atari environment, Simulator segment is used to define the part of the model which handles the objects related to the different simulators individually catering to the environment of the specific game. The train-Atari segment is the part of our model which is actually involved with the training process where the machine learns to play the game.

e.) Use Phase

After the implementation phase we move on to the use phase where we have trained our model to train and play two games, Pong and Space Invader perfectly achieving almost perfect score every time. These use phases were used to test the model we can further extend our model to train different environments and use them in different classes of Applications such as Robotics, Finance Sector, Medical Sector (Eg. Amblyopia)

6.2 SYSTEM DESIGN

6.2.1 Low Level Design

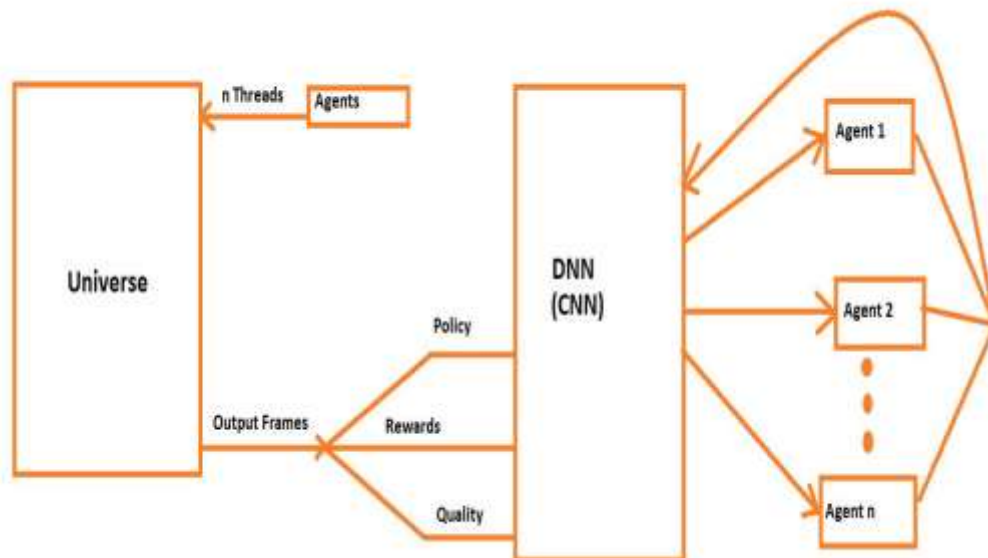


Fig 6: Low Level System Design

6.2.2 High Level Design

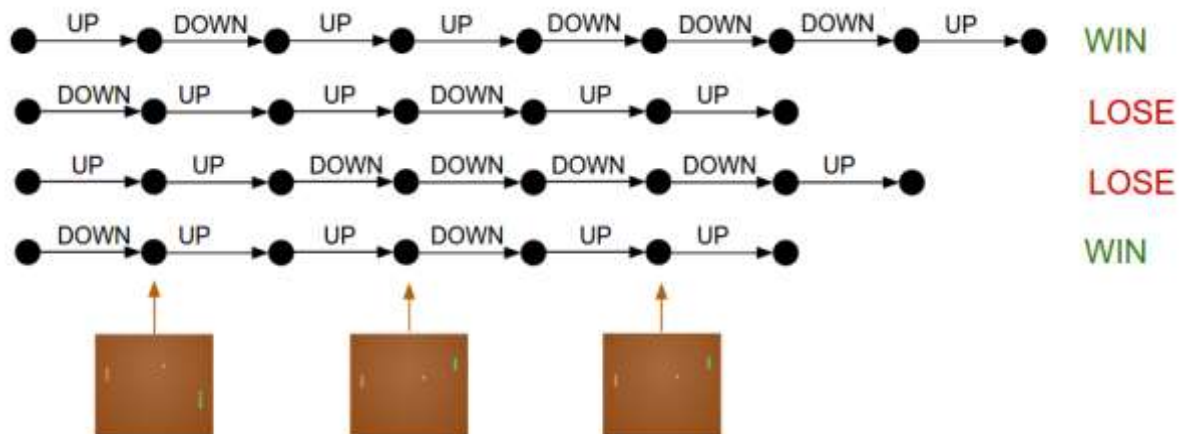


Fig 7: High Level Design

CHAPTER 7

IMPLEMENTATION

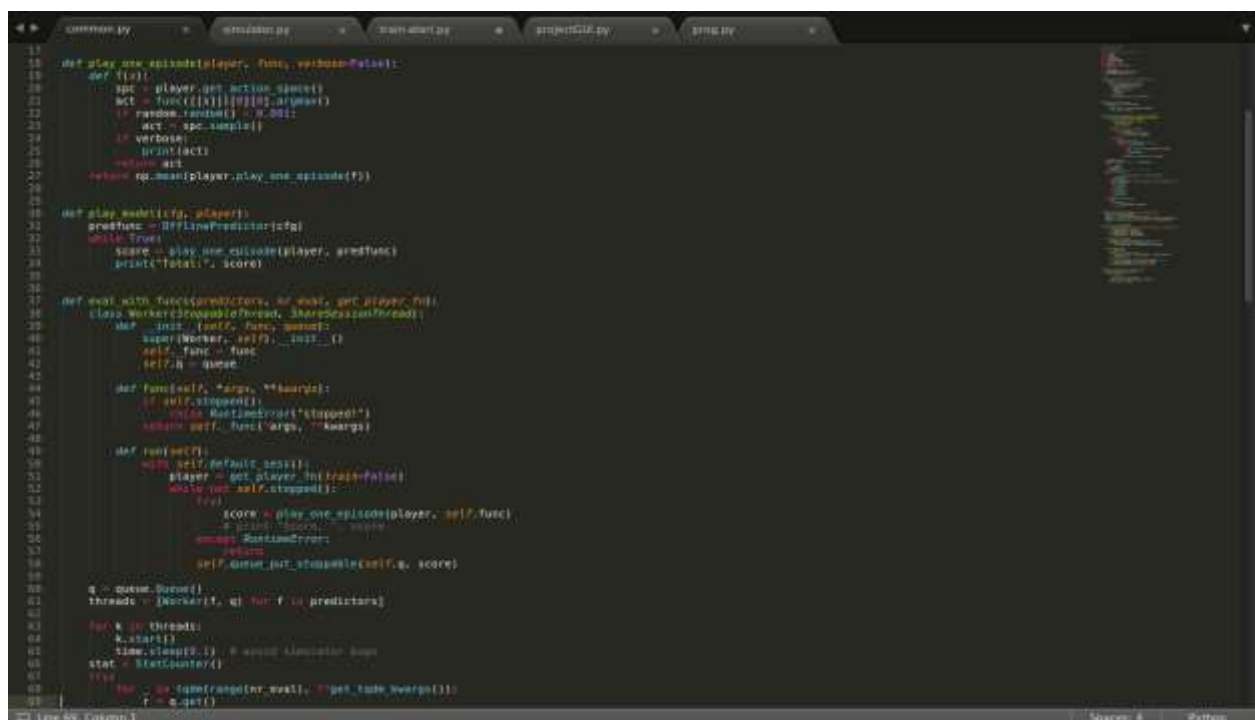
CHAPTER 7

IMPLEMENTATION

7.1 IMPLEMENTATION

The design carried out in order to develop the model for being able to train and test and imply reinforcement learning was translated into a working model by implementing our design process which was broken down into three segments of programmed source code. Each of the three segments were responsible for handling an aspect of the working model. We have described the working of each of the segments along with the code snippets and implementation in the following subsections.

7.1.1 Common.py



```

11
12 def play_one_episode(player, func, verbose=False):
13     def test():
14         player.get_action_space()
15         act = func([x] * [0] * [0] * [0])
16         if random.random() < 0.01:
17             act = spc.sample()
18         verbose = True
19         print(act)
20         return act
21     return np.mean(player.play_one_episode(func))
22
23 def play_model(cfg, player):
24     predfunc = OfflinePredictor(cfg)
25     while True:
26         score = play_one_episode(player, predfunc)
27         print("total", score)
28
29 def test_with_func(predfunc, or_test, get_player_func):
30     class Worker(Thread, SharedObjectThread):
31         def __init__(self, func, queue):
32             super(Worker, self).__init__()
33             self.func = func
34             self.q = queue
35
36         def run(self, *args, **kwargs):
37             if self.stopped:
38                 return
39             while self.stopped:
40                 self.q.put(args, **kwargs)
41
42         def run(self):
43             while self.stopped:
44                 player = get_player(cfg)
45                 while not self.stopped:
46                     score = play_one_episode(player, self.func)
47                     self.q.put(score)
48                     self.stopped = True
49
50     q = queue.Queue()
51     threads = [Worker(f, q) for f in predfuncs]
52
53     for t in threads:
54         t.start()
55     time.sleep(1) # wait for all threads to start
56     stat = StatCounter()
57     for i in range(nr_walls, 0, -1):
58         for t in threads:
59             t = q.get()
60

```

This part of the code gives us generic handles to execute common functions like play, test or train.

7.1.2 Simulator

```

30
31 class TransactionException(Exception):
32     """A transaction of state, or experience"""
33
34     def __init__(self, state, action, reward, **kwargs):
35         """A transaction of state, or experience"""
36         self.state = state
37         self.action = action
38         self.reward = reward
39         self.kwargs = kwargs
40         self.__dict__ = self.kwargs
41
42
43 class SimulatorProcessBase(ABC):
44     """Base class for simulator processes"""
45     def __init__(self, id):
46         super(SimulatorProcessBase, self).__init__()
47         self.id = id
48         self.name = f'simulator-{id}'
49         self.identity = self.name.encode('utf-8')
50
51     @abstractmethod
52     def build_player(self):
53         """Build a player"""
54
55
56 class SimulatorProcessStateExchange(SimulatorProcessBase):
57     """A process that simulates a player and communicates to another to
58     send states and receive the next action"""
59
60     def __init__(self, c2s, s2c, pipe_c2s, pipe_s2c):
61         """Init"""
62         super(SimulatorProcessStateExchange, self).__init__(id)
63         self.c2s = pipe_c2s
64         self.s2c = pipe_s2c
65
66     def run(self):
67         player = self.build_player()
68         context = zmq.Context()
69         c2s_socket = context.socket(zmq.PUSH)
70         c2s_socket.setsockopt(zmq.IDENTITY, self.identity)
71         c2s_socket.set_hwm(1)
72         c2s_socket.connect(self.c2s)
73
74         s2c_socket = context.socket(zmq.DEALER)
75         s2c_socket.setsockopt(zmq.IDENTITY, self.identity)
76         s2c_socket.connect(self.s2c)
77
78         while True:
79             # Receive state from c2s
80             state = self.c2s.recv_multipart()
81             # Send action to s2c
82             action = self.s2c.send_multipart(state)

```

This code is the simulation code.

7.1.3 Hyper-parameters

```

1  # Hyper-parameters
2  # =====
3  # =====
4  # =====
5  # =====
6  # =====
7  # =====
8  # =====
9  # =====
10 # =====
11 # =====
12 # =====
13 # =====
14 # =====
15 # =====
16 # =====
17 # =====
18 # =====
19 # =====
20 # =====
21 # =====
22 # =====
23 # =====
24 # =====
25 # =====
26 # =====
27 # =====
28 # =====
29 # =====
30 # =====
31 # =====
32 # =====
33 # =====
34 # =====
35 # =====
36 # =====
37 # =====
38 # =====
39 # =====
40 # =====
41 # =====
42 # =====
43 # =====
44 # =====
45 # =====
46 # =====
47 # =====
48 # =====
49 # =====
50 # =====
51 # =====
52 # =====
53 # =====
54 # =====
55 # =====
56 # =====
57 # =====
58 # =====
59 # =====
60 # =====
61 # =====
62 # =====
63 # =====
64 # =====
65 # =====
66 # =====
67 # =====
68 # =====
69 # =====
70 # =====
71 # =====
72 # =====
73 # =====
74 # =====
75 # =====
76 # =====
77 # =====
78 # =====
79 # =====
80 # =====
81 # =====
82 # =====
83 # =====
84 # =====
85 # =====
86 # =====
87 # =====
88 # =====
89 # =====
90 # =====
91 # =====
92 # =====
93 # =====
94 # =====
95 # =====
96 # =====
97 # =====
98 # =====
99 # =====
100 # =====

```

7.1.4 Graph and Simulation Statistics

```
class MySimulatorMaster(SimulatorMaster, Callback):
    def __init__(self, pipe_c2s, pipe_s2c, model):
        super(MySimulatorMaster, self).__init__(pipe_c2s, pipe_s2c)
        self.M = model
        self.queue = queue.Queue(maxsize=BATCH_SIZE * 8 * 2)

    def setup_graph(self):
        self.async_predictor = MultiThreadAsyncPredictor(
            self.trainer.get_predictors(['state'], ['policy_explore', 'pred_value'],
                                       PREDICTOR_THREAD, batch_size=PREDICT_BATCH_SIZE)

    def before_train(self):
        self.async_predictor.start()

    def on_state(self, state, ident):
        def cb(outputs):
            try:
                distrib, value = outputs.result()
            except CanceledError:
                logger.info("Client {} cancelled.".format(ident))
                return
            assert np.all(np.isfinite(distrib)), distrib
            action = np.random.choice(len(distrib), p=distrib)
            client = self.clients[ident]
            client.memory.append(TransitionExperience(state, action, None, value=value))
            self.send_queue.put([ident, dumps(action)])
            self.async_predictor.put_task([state], cb)

    def on_episode_over(self, ident):
        self.parse_memory(0, ident, True)

    def on_datapoint(self, ident):
        client = self.clients[ident]
        if len(client.memory) == LOCAL_TIME_MAX + 1:
            R = client.memory[-1].value
            self.parse_memory(R, ident, False)

    def parse_memory(self, init_r, ident, isOver):
        client = self.clients[ident]
        mem = client.memory
        if not isOver:
            last = mem[-1]
            mem = mem[:-1]

        mem.reverse()
        R = float(init_r)
        for idx, k in enumerate(mem):
            R = np.clip(k.reward, -1, 1) + GAMMA * R
            self.queue.put([k.state, k.action, R])

        if not isOver:
            client.memory = [last]
```

7.2 EXECUTION PHASE

The implementation of the code results in three programs, which we use to train our input models which were initially stored in the form of sparse matrix and then the final part of the implementation phase consists of the module which was developed to run the test cases and test the models which was trained. We made use of the tKinter library in Python to develop the interface for user interaction with the model system to run and test the models.

```
from Tkinter import *
import os
root = Tk()
def Untrained(event):
    print "Un"
def Trained(event):
    print "Tra"
def Trained100(event):
    print "100"
def Coaster(event):
    print "Coas"
    os.system("python train-atari.py --task play --env SpaceInvaders-v0 --load SpaceInvaders-v0.tfmodel")
def Pong(event):
    print "Coas"
    os.system("python train-atari.py --task play --env Pong-v0 --load Pong-v0.tfmodel")
frame = Frame(root)
```



```
frame.grid()
Label1 = Label(root, text = "Game of Pong")
Label1.grid(row = 0, column = 2)
#Label1.pack(side = LEFT)
photo1 = PhotoImage(file = 'pong.png')
label2 = Label(root, image = photo1)
label2.grid(row = 1, column = 2)

button1 = Button(root, text = "Untrained", fg = "red",)
button1.grid(row = 2, column = 0,)
button1.bind("<Button-1>", Untrained)
#button1.pack(side = BOTTOM)
button2 = Button(root, text = "Trained", fg = "blue")
button2.grid(row = 2, column = 2, padx = 9)
button2.bind("<Button-1>", Trained)

button3 = Button(root, text = "100% Trained", fg = "green")
button3.grid(row = 2, column = 5)
button3.bind("<Button-1>", Trained100)

Label3 = Label(root, text = "Game of Coaster Racer")
Label3.grid(row = 10, column = 2)

photo2 = PhotoImage(file = 'coas.png')
label4 = Label(root, image = photo2)
label4.grid(row = 11, column = 2)

button4 = Button(root, text = "PLAY SPACE INVADER", fg = "green")
button4.grid(row = 12, column = 2)
button4.bind("<Button-1>", Coaster)

button4 = Button(root, text = "PLAY PONG", fg = "green")
button4.grid(row = 12, column = 4)
button4.bind("<Button-1>", Pong)
#frame.pack()
root.minsize(320,300)
root.mainloop()
```

We use the tKinter library in Python to develop the interface for user interaction with the model system to run and test the models.

CHAPTER 8

EXECUTION AND TEST CASES

CHAPTER 8

EXECUTION AND TEST CASES

8.1 EXECUTION

The User Interface which is displayed for the user to interact with the working, completely trained, untrained and partially trained models running in the system.



Fig 8: Snapshot of the GUI

8.1.1 Test Runs

We have captured the screenshots of the test runs performed on two games as shown below.

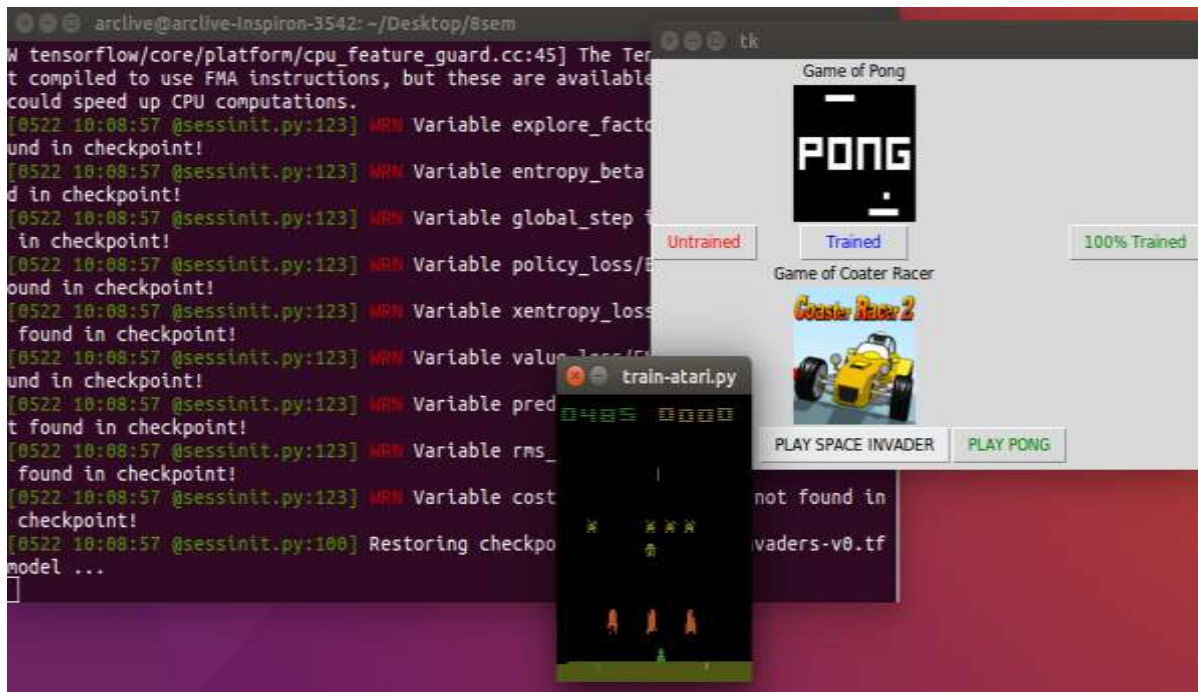


Fig 9: Snap of the final testing phase of the Space Invader Game

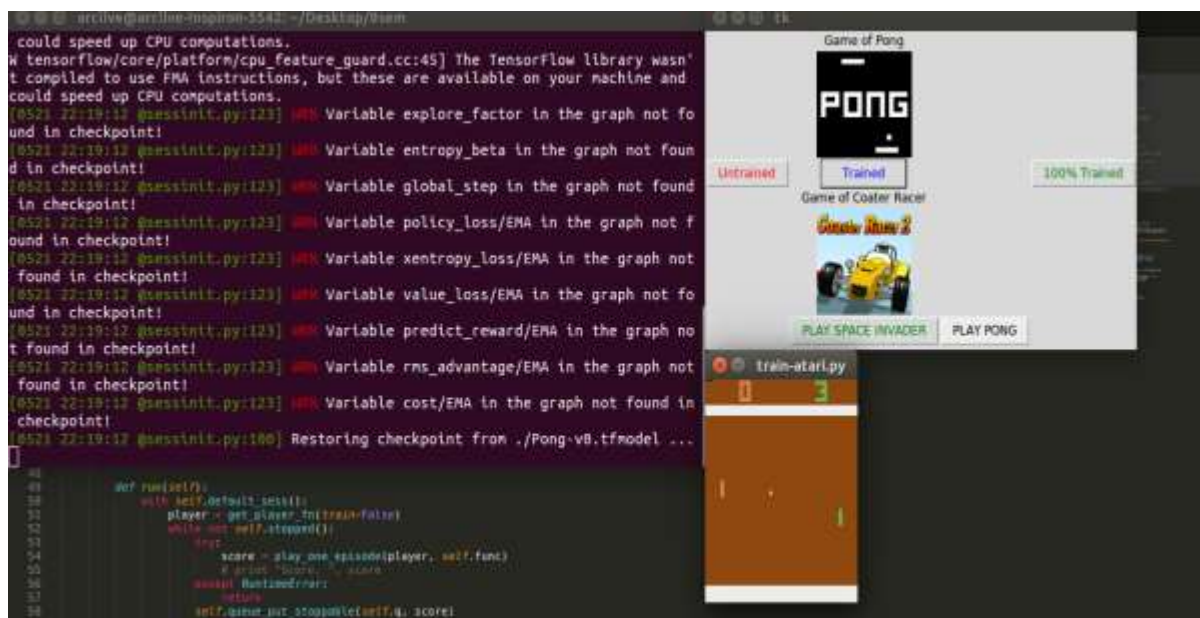


Fig 10: Snap of the final testing phase of the Pong Game

8.1.2 Observations and Graphs

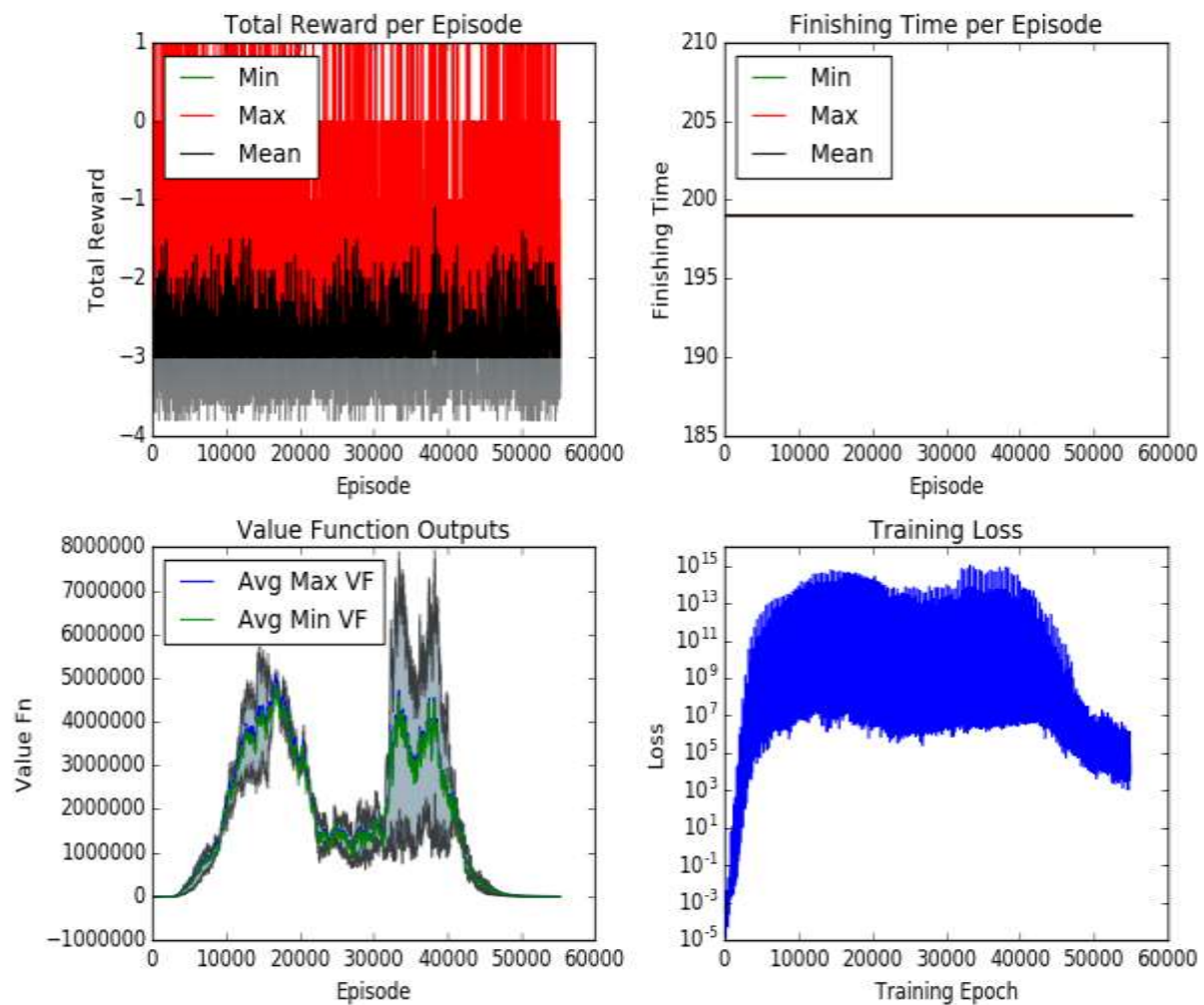


Fig 11: Quality Graphs

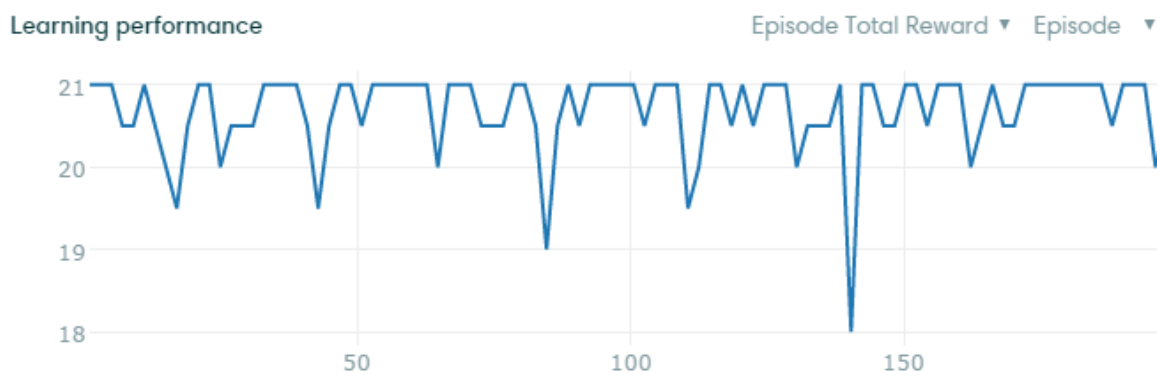


Fig 12: Runtime Graph

CHAPTER 9

CONCLUSION AND FUTURE ENHANCEMENTS

CHAPTER 9

CONCLUSION AND FUTURE ENCHANCEMENTS

9.1. CONCLUSION

Unlike policy gradient methods, which attempt to learn functions which directly map an observation to an action, Q-Learning attempts to learn the value of being in a given state, and taking a specific action there.

This is a difficult problem due to four aspects.

1. Exploration vs Exploitation
2. Stochasticity of taking Actions
3. Conditional Rewards
4. Delayed Rewards

We try to mitigate the problem of exploration vs exploitation, and solve it efficiently with a modification of Deep Q Network algorithm. Our main modifications are:

1. Using OpenAI Gym to get the environment.
2. Using Gym helped making the code environment agnostic.
3. Parallelizing DQN.

9.2 FUTURE WORK

Future work includes finding the best tradeoff b/w gradient descent and number of parallel workers.

9.3 PROJECT ACTIVITY

Project Activity Gantt Chart

This project is completed in a total duration of three months, along with three monthly project reviews which were conducted in the month of March, April and May.




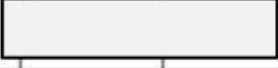



Development Phase	90 Days						Duration (Day)
	0 to 15 Day	16 to 30 Day	31 to 45 Day	46 to 60 Day	61 to 75 Day	76 to 90 Day	
Requirement Gathering							10
Analysis							15
Design							30
Coding							25
Testing							12
Implementation							08
Documentation							80
Total Time (Days)							90

Fig 13: Gantt chart

REFERENCES

REFERENCES

- [1] Hierarchical Deep Reinforcement Learning: Integrating Temporal Abstraction and Intrinsic Motivation by Tejas D. Kulkarni, Karthik R. Narasimhan, Ardavan Saeedi and Joshua B. Tenenbaum
- [2] Bellemare, M. G., Ostrovski, G., Guez, A., Thomas, P. S., and Munos, R. (2016). Increasing the action gap: New operators for reinforcement learning. In Proceedings of the 30th AAAI Conference on Artificial Intelligence.
- [3] Bellemare, M. G., Naddaf, Y., Veness, J., and Bowling, M. (2013). The Arcade Learning Environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 47:253–279.
- [4] Bellemare, M. G., Veness, J., and Bowling, M. (2012). Investigating contingency awareness using Atari 2600 games. In Proceedings of the 26th AAAI Conference on Artificial Intelligence.
- [5] Jaksch, T., Ortner, R., and Auer, P. (2010). Near-optimal regret bounds for reinforcement learning. *Journal of Machine Learning Research*, 11:1563–1600.
- [6] Diuk, C., Cohen, A., and Littman, M. L. (2008). An object-oriented representation for efficient reinforcement learning. In Proceedings of the 25th International Conference on Machine Learning, pages 240–247. ACM.
- [7] Even-Dar, E. and Mansour, Y. (2001). Convergence of optimistic and incremental Q-learning. In *Advances in Neural Information Processing Systems 14*.
- [8] A. McGovern and A. G. Barto. Automatic discovery of subgoals in reinforcement learning using diverse density. *Computer Science Department Faculty Publication Series*, page 8, 2001.
- [9] Dearden, R., Friedman, N., and Russell, S. (1998). Bayesian Q-learning. In Proceedings of the Fifteenth National Conference on Artificial Intelligence, pages 761–768.