## Declaration

We hereby declare that the project entitled **"Asynchronous method for deep reinforcement learning"** submitted by us to Sri Jayachamarajendra College of Engineering, Mysore in fulfillment of the requirement for the award of the degree of Bachelor of Engineering in Computer Science Department is a record of bonafide project work carried out by us under the guidance of Prof.Ashritha R Murthy. We further declare that the work reported in this project has not been submitted, either in part or in full, for the award of any other degree or diploma in this institute or any other institute or university.

Nithin Prabhu G
4JC15CS068

Shwetha U
4JC16CS419

Yathish G
4JC16CS422

**Abstract**

The project proposes a conceptually simple and lightweight framework for deep reinforcement learning that uses asynchronous gradient descent for optimization of deep neural network controllers. It also presents asynchronous variants of four standard reinforcement learning algorithm and show that 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[11]. Furthermore, it is shown that asynchronous actor-critic succeeds on a wide variety of continuous motor control problems as well as on a new task of navigating random 3D mazes using a visual input.

**Acknowledgement**

It gives us immense pleasure to write an acknowledgement to this project, a contribution of all the people who helped to realize it. We extend our deep regards to Dr.T.N.Nagabhushan, Honorable Principal of Sri Jayachamarajendra College of Engineering, for providing an excellent environment for our education and his encouragement throughout our stay in college. We would like to convey our heartfelt thanks to our HOD, Dr. M.P.Pushpalatha, for giving us the opportunity to embark upon this topic. We would like to thank our project guide, Prof.Ashritha R Murthy for their invaluable guidance and enthusiastic assistance and for providing us support and constructive suggestions for the betterment of the project, without which this project would not have been possible. We appreciate the timely help and kind cooperation of our lecturers, other staff members of the department and our seniors, with whom we have come up all the way during our project work without whose support this project would not have been success. Finally, we would like to thank our friends for providing numerous insightful suggestions. We also convey our sincere thanks to all those who have contributed to this learning opportunity at every step of this project.

# Contents

# Chapter 1

# Introduction

## 1.1 Introduction to the problem domain

Deep neural networks provide rich representations that can enable reinforcement learning (RL) algorithms to perform effectively. However, it was previously thought that the combination of simple online RL algorithms with deep neural networks was fundamentally unstable. Instead, a variety of solutions have been proposed to stabilize the algorithm[7]. These approaches share a common idea: the sequence of observed data encountered by an online RL agent is non-stationary, and on-line RL updates are strongly correlated. By storing the agents data in an experience replay memory, the data can be batched or randomly sampled from different time-steps[15]. Aggregating over memory in this way reduces non-stationarity and decorrelates updates, but at the same time limits the methods to off-policy reinforcement learning algorithms.

Deep RL algorithms based on experience replay have achieved unprecedented success in challenging domains such as Atari 2600. However, experience replay has several drawbacks: it uses more memory and computation per real interaction; and it requires off-policy learning algorithms that can update from data generated by an older policy.

The project provides a very different paradigm for deep reinforcement learning. Instead of experience replay, we asynchronously execute multiple agents in parallel, on multiple instances of the environment. This parallelism also decorrelates the agents data into a more stationary process, since at any given time-step the parallel agents will be experiencing a variety of different states. This simple idea enables a much larger spectrum of fundamental on-policy RL algorithms, such as Sarsa, n-step methods, and actor critic methods, as well as off-policy RL algorithms such as Q-learning, to be applied

robustly and effectively using deep neural networks.

Our parallel reinforcement learning paradigm also offers practical benefits. Whereas previous approaches to deep reinforcement learning rely heavily on specialized hardware such as GPUs or massively distributed architectures , our experiments run on a single machine with a standard multi-core CPU. When applied to a variety of Atari 2600 domains, on many games asynchronous reinforcement learning achieves better results, in far less time than previous GPU-based algorithms, using far less resource than massively distributed approaches. The best of the proposed methods, asynchronous advantage actor critic (A3C), also mastered a variety of continuous motor control tasks as well as learned general strategies for exploring 3D mazes purely from visual inputs. It is believed that the success of A3C on both 2D and 3D games, discrete and continuous action spaces, as well as its ability to train feed forward and recurrent agents makes it the most general and successful reinforcement learning agent to date[3].

## 1.2 Aim/Statement of the Problem

The project proposes a conceptually simple and lightweight framework for deep reinforcement learning that uses asynchronous gradient descent for optimization of deep neural network controllers.

## 1.3 Objectives of the Project Work

- The project provides a very different paradigm for deep reinforcement learning. Instead of experience replay, we asynchronously execute multiple agents in parallel, on multiple instances of the environment.
- The project incorporates parallelism which also decorrelates the agents data into a more stationary process, since at any given time-step the parallel agents will be experiencing a variety of different states.
- The core objective is to enable a much larger spectrum of fundamental on-policy RL algorithms, such as Sarsa, n-step methods, and actor critic methods, as well as off-policy RL algorithms such as Q-learning, to be applied robustly and effectively using deep neural networks.

## 1.4 Applications

- Natural Language Processing which is used heavily in language conversion in chat rooms or processing text from where human speeches.
- Optical Character Recognition which is scanning of images. It's gaining traction lately to read an image and extract text out of it and correlate to the objects found on image.
- Speech Recognition applications like Siri or Cortana needs no introduction.
- Artificial Intelligence induction to different robots for automating at least a minute level of tasks a human can do that can be a little smarter.
- Drug discovery though medical imaging-based diagnosis using deep learning. It's kind of in early stages now. Check Butterfly Network for the work they are doing.
- CRM needs for companies are growing day by day. There are hundreds of thousands of companies around the globe from small to big companies who wants to know their potential customers. Deep Learning has provided some outstanding results. Check for companies like RelateIQ(Product) who has seen astounding success of using Machine Learning in this area.

## 1.5 Existing solution method

The General Reinforcement Learning Architecture (Gorila) of performs asynchronous training of reinforcement learning agents in a distributed setting. In Gorila, each process contains an actor that acts in its own copy of the environment, a separate replay memory, and a learner that samples data from the replay memory and computes gradients of the DQN loss with respect to the policy parameters. The gradients are asynchronously sent to a central parameter server which updates a central copy of the model. The updated policy parameters are sent to the actor-learners at fixed intervals. By using 100 separate actor-learner processes and 30 parameter server instances, a total of 130 machines, Gorila was able to significantly outperform DQN over 49 Atari games. On many games Gorila reached the score achieved by DQN over 20 times faster than DQN. It is to be noted that a similar way of parallelizing DQN was proposed by Chavez.

In earlier work, applied the Map Reduce framework to parallelizing batch reinforcement learning methods with linear function approximation. Parallelism was used to speed up large matrix operations but not to parallelize the collection of experience or stabilize learning. Grounds & Kudenko, proposed a parallel version of the Sarsa algorithm that uses multiple separate actor-learners to accelerate training. Each actorlearner learns separately and periodically sends updates to weights that have changed significantly to the other learners using peer-to-peer communication[10].

Tsitsiklis studied convergence properties of Qlearning in the asynchronous optimization setting. These results show that Q-learning is still guaranteed to converge when some of the information is outdated as long as outdated information is always eventually discarded and several other technical assumptions are satisfied. Even earlier, studied the related problem of distributed dynamic programming.

Another related area of work is in evolutionary methods, which are often straightforward to parallelize by distributing fitness evaluations over multiple machines or threads . Such parallel evolutionary apapproaches have recently been applied to some visual reinforcement learning tasks. In one example,evolved convolutional neural network controllers for the TORCS driving simulator by performing fitness evaluations on 8 CPU cores in parallel.

### 1.5.1 Asynchronous Method

The study considers the backdrop of the standard RL method for developing asynchronous algorithms. For creating an RL framework, the researchers follow a two-step approach. First, they use a technique called asynchronous actor-learners, due to its robustness. For this, they use a single machine with multiple CPU threads, mainly to lower communication costs between threads and achieve efficient algorithm updates. Second, they analyse the various actor-learners present and use exploration policies on these learners. This provides the advantage of applying online updates parallelly with better correlation in the algorithm. In addition, these parallel multiple actor learners have manifold benefits such as reducing training time and promoting online RL.

## 1.6 Proposed solution method

### 1.6.1 Asynchronous RL Framework

The project now presents multi-threaded asynchronous variants of one-step Sarsa, one-step Q-learning, n-step Q-learning, and advantage actor-critic. The aim in designing these methods was to find RL algorithms that can train deep neural network policies reliably and without large resource requirements. While the underlying RL methods are quite different, with actor-critic being an on-policy policy search method and Q-learning being an off-policy value-based method, we use two main ideas to make all four algorithms practical given our design goal.

First, we use asynchronous actor-learners, similarly to the Gorila framework , but instead of using separate machines and a parameter server, we use multiple CPU threads on a single machine. Keeping the learners on a single machine removes the communication costs of sending gradients and parameters and enables us to use Hogwild! style updates for training[9].

Second, we make the observation that multiple actors-learners running in parallel are likely to be exploring different parts of the environment. Moreover, one can explicitly use different exploration policies in each actor-learner to maximize this diversity. By running different exploration policies in different threads, the overall changes being made to the parameters by multiple actor-learners applying online updates in parallel are likely to be less correlated in time than a single agent applying online updates. Hence, we do not use a replay memory and rely on parallel actors employing different exploration policies to perform the stabilizing role undertaken by experience replay in the DQN training algorithm.

In addition to stabilizing learning, using multiple parallel actor-learners has multiple practical benefits. First, we obtain a reduction in training time that is roughly linear in the number of parallel actor-learners. Second, since we no longer rely on experience replay for stabilizing learning we are able to use on-policy reinforcement learning methods such as Sarsa and actor-critic to train neural networks in a stable way. The project now describes the variants of one-step Qlearning, one-step Sarsa, n-step Q-learning and advantage actor-critic.

### 1.6.2 Asynchronous one-step Q-learning

Pseudocode for our variant of Q-learning, which we call Asynchronous onestep Q-learning, is used in the initial step. Each thread interacts with its own copy of the environment and at each step computes a gradient of the Q-learning loss. The project uses a shared and slowly changing target network in computing the Q-learning loss, as was proposed in the DQN training method. The project also accumulates gradients over multiple timesteps before they are applied, which is similar to using minibatches. This reduces the chances of multiple actor learners overwriting each others updates. Accumulating updates over several steps also provides some ability to trade off computational efficiency for data efficiency.

Finally, we found that giving each thread a different exploration policy helps improve robustness. Adding diversity to exploration in this manner also generally improves performance through better exploration. While there are many possible ways of making the exploration policies differ we experiment with using epsilion greedy exploration with epsilion periodically sampled from some distribution by each thread. However this method was found to be ineffecient and therefore we follow n-steps Q-learning.

### 1.6.3 Asynchronous Advantage actor-critic

The A3C algorithm was released by Googles DeepMind group earlier this year, and it made a splash by essentially obsoleting DQN. It was faster, simpler, more robust, and able to achieve much better scores on the standard battery of Deep RL tasks. On top of all that it could work in continuous as well as discrete action spaces. Given this, it has become the go-to Deep RL algorithm for new challenging problems with complex state and action spaces. In fact, OpenAI just released a version of A3C as their universal starter agent for working with their new (and very diverse) set of Universe environments[13].

#### 1.6.3.1 Asynchronous

Unlike DQN, where a single agent represented by a single neural network interacts with a single environment, A3C utilizes multiple incarnations of the above in order to learn more efficiently. In A3C there is a global network, and multiple worker agents which each have their own set of network parameters. Each of these agents interacts with its own copy of the environment at the same time as the other agents are interacting with their environments.

The reason this works better than having a single agent (beyond the speedup of getting more work done), is that the experience of each agent is independent of the experience of the others. In this way the overall experience available for training becomes more diverse.

#### 1.6.3.2 Advantage

If we think back to our implementation of Policy Gradient, the update rule used the discounted returns from a set of experiences in order to tell the agent which of its actions were good and which were bad. The network was then updated in order to encourage and discourage actions appropriately.

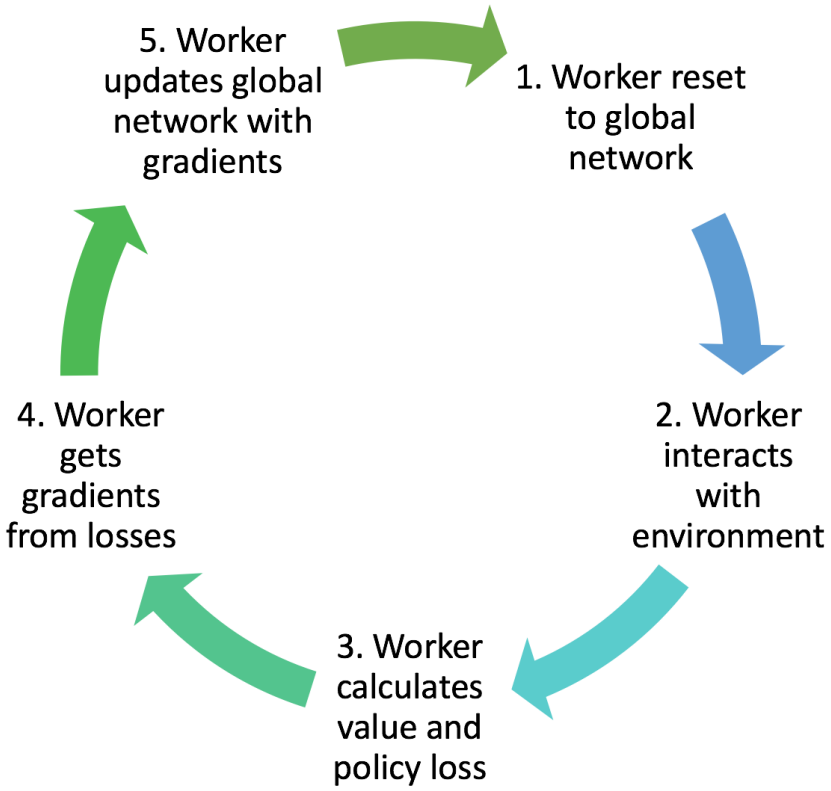


Figure 1.1: Training workflow of each worker agent in A3C.

#### 1.6.3.3 Actor-Critic

So far this series has focused on value-iteration methods such as Q-learning, or policy-iteration methods such as Policy Gradient. Actor-Critic combines the benefits of both approaches. In the case of A3C, our network will estimate both a value function V(s) (how good a certain state is to be in) and a policy (s) (a set of action probability outputs). These will each be separate fully-connected layers sitting at the top of the network. Critically, the agent uses the value estimate (the critic) to update the policy (the actor) more intelligently than traditional policy gradient methods.

### 1.6.4 Optimization

Three different optimization algorithms are investigated in our asynchronous framework SGD with momentum, RMSProp (Tieleman and Hinton, 2012) without shared statistics, and RMSProp with shared statistics. The project uses the standard non-centered RMSProp update which is moere effecient anh helps to achieve higher optimization when compared to others. A comparison on a subset of Atari 2600 games showed that a variant of RMSProp where statistics g are shared across threads is considerably more robust than the other two methods.

#### 1.6.4.1 RMSProp

While RMSProp (Tieleman and Hinton, 2012) has been widely used in the deep learning literature, it has not been extensively studied in the asynchronous optimization setting. The standard non-centered RMSProp update is used in our project to achieve higher optimization values.

In order to apply RMSProp in the asynchronous optimization setting one must decide whether the moving average of elementwise squared gradients g is shared or per-thread. The project is experimented with two versions of the algorithm. In one version, which is referred to as RMSProp, each thread maintains its own g value. In the other version, which is called Shared RMSProp, the vector g is shared among threads and is updated asynchronously and without locking. Sharing statistics among threads also reduces memory requirements by using one fewer copy of the parameter vector per thread.

The project is compared with these three asynchronous optimization algorithms in terms of their sensitivity to different learning rates and random network initializations. Figure 1.2 shows a comparison of the methods for two different reinforcement learning methods (Async n-step Q and Async Advantage Actor-Critic) on four different games (Breakout, Beamrider, Seaquest

and Space Invaders). Each curve shows the scores for 50 experiments that correspond to 50 different random learning rates and initializations. The x-axis shows the rank of the model after sorting in descending order by final average score and the y-axis shows the final average score achieved by the corresponding model. In this representation, the algorithm that performs better would achieve higher maximum rewards on the y-axis and the algorithm that is most robust would have its slope closest to horizontal, thus maximizing the area under the curve. RMSProp with shared statistics tends to be more robust than RMSProp with per-thread statistics, which is in turn more robust than Momentum SGD.

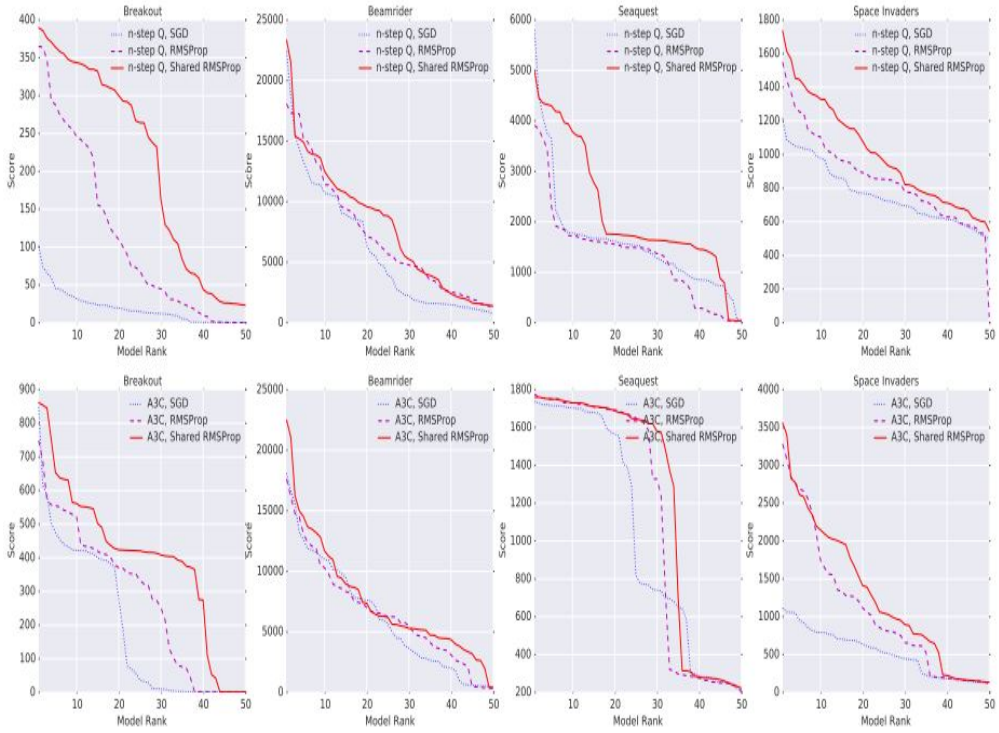


Figure 1.2: Comparison of three different optimization methods (Momentum SGD, RMSProp, Shared RMSProp) tested using two different algorithms (Async n-step Q and Async Advantage Actor-Critic)

## 1.7 Time schedule for completion of the project work (Gantt chart)

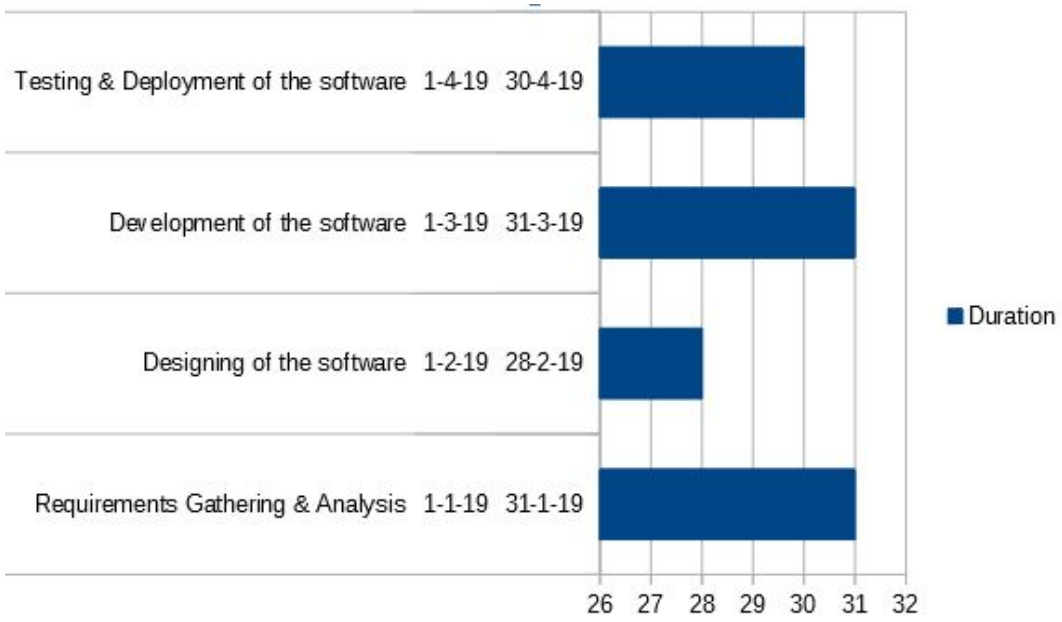


Figure 1.3: Gantt chart

# Chapter 2

# Literature Survey

The General Reinforcement Learning Architecture (Gorila) of (Nair et al., 2015) performs asynchronous training of reinforcement learning agents in a distributed setting. In Gorila, each process contains an actor that acts in its own copy of the environment, a separate replay memory, and a learner that samples data from the replay memory and computes gradients of the DQN loss (Mnih et al., 2015) with respect to the policy parameters. The gradients are asynchronously sent to a central parameter server which updates a central copy of the model. The updated policy parameters are sent to the actor-learners at fixed intervals. By using 100 separate actor-learner processes and 30 parameter server instances, a total of 130 machines, Gorila was able to significantly outperform DQN over 49 Atari games. On many games Gorila reached the score achieved by DQN over 20 times faster than DQN. It is also noted that a similar way of parallelizing DQN was proposed by Chavez et al., 2015[2].

In earlier work, (Li & Schuurmans, 2011) applied the Map Reduce framework to parallelizing batch reinforcement learning methods with linear function approximation. Parallelism was used to speed up large matrix operations but not to parallelize the collection of experience or stabilize learning. (Grounds & Kudenko, 2008) proposed a parallel version of the Sarsa algorithm that uses multiple separate actor-learners to accelerate training. Each actorlearner learns separately and periodically sends updates to weights that have changed significantly to the other learners using peer-to-peer communication.

(Tsitsiklis, 1994) studied convergence properties of Q learning in the asynchronous optimization setting. These results show that Q-learning is still guaranteed to converge when some of the information is outdated as long as outdated information is always eventually discarded and several other technical assumptions are satisfied. Even earlier, (Bertsekas, 1982) studied the related problem of distributed dynamic programming.

Another related area of work is in evolutionary methods, which are often straightforward to parallelize by distributing fitness evaluations over multiple machines or threads (Tomassini, 1999)[14]. Such parallel evolutionary approaches have recently been applied to some visual reinforcement learning tasks. In one example, (Koutnk et al., 2014)[5] evolved convolutional neural network controllers for the TORCS driving simulator by performing fitness evaluations on 8 CPU cores in parallel.

In the work of Human-level control through deep reinforcement learning, the authors Mnih Volodymyr, Koray Kavukcuoglu,etc convey that the theory of reinforcement learning provides a normative account, deeply rooted in psychological and neuroscientific perspectives on animal behaviour, of how agents may optimize their control of an environment. To use reinforcement learning successfully in situations approaching real-world complexity, however, agents are confronted with a difficult task: they must derive efficient representations of the environment from high-dimensional sensory inputs, and use these to generalize past experience to new situations. Remarkably, humans and other animals seem to solve this problem through a harmonious combination of reinforcement learning and hierarchical sensory processing systems, the former evidenced by a wealth of neural data revealing notable parallels between the phasic signals emitted by dopaminergic neurons and temporal difference reinforcement learning algorithms.

While reinforcement learning agents have achieved some successes in a variety of domains, their applicability has previously been limited to domains in which useful features can be handcrafted, or to domains with fully observed, low-dimensional state spaces. Here we use recent advances in training deep neural networks to develop a novel artificial agent, termed a deep Q-network, that can learn successful policies directly from high-dimensional sensory inputs using end-to-end reinforcement learning. The project is tested with this agent on the challenging domain of classic Atari 2600 games. The project demonstrates that the deep Q-network agent, receiving only the pixels and the game score as inputs, was able to surpass the performance of all previous algorithms and achieve a level comparable to that of a professional human games tester across a set of 49 games, using the same algorithm, network architecture and hyperparameters.

This work bridges the divide between high-dimensional sensory inputs and actions, resulting in the first artificial agent that is capable of learning to excel at a diverse array of challenging tasks.

In the work Playing Atari with Deep Reinforcement Learning, the authors Volodymyr Mnih, Koray Kavukcuoglu, etc have presented the first deep learning model 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. They apply our method to seven Atari 2600 games from the Arcade Learning Environment[1], with no adjustment of the architecture or learning algorithm. They also find that it outperforms all previous approaches on six of the games and surpasses a human expert on three of them.

Perhaps the most similar prior work to the own approach is neural fitted Q-learning (NFQ). NFQ optimises the sequence of loss functions, using the RPROP algorithm to update the parameters of the Q-network. However, it uses a batch update that has a computational cost per iteration that is proportional to the size of the data set, whereas we consider stochastic gradient updates that have a low constant cost per iteration and scale to large data-sets. NFQ has also been successfully applied to simple real-world control tasks using purely visual input, by first using deep autoencoders to learn a low dimensional representation of the task, and then applying NFQ to this representation. In contrast our approach applies reinforcement learning end-to-end, directly from the visual inputs; as a result it may learn features that are directly relevant to discriminating action-values. Q-learning has also previously been combined with experience replay and a simple neural network, but again starting with a low-dimensional state rather than raw visual inputs.

The use of the Atari 2600 emulator as a reinforcement learning platform was introduced in which applied standard reinforcement learning algorithms with linear function approximation and generic visual features. Subsequently, results were improved by using a larger number of features, and using tug-of-war hashing to randomly project the features into a lower-dimensional space. The HyperNEAT evolutionary architecture has also been applied to the Atari platform, where it was used to evolve (separately, for each distinct game) a neural network representing a strategy for that game. When trained repeatedly against deterministic sequences using the emulators reset facility, these strategies were able to exploit design flaws in several Atari games.

In the work Deep Reinforcement Learning for Flappy Bird by Kevin Chen, the author conveys that reinforcement learning is essential for applications where there is no single correct way to solve a problem. In this project, he shows that deep reinforcement learning is very effective at learning how to play the game Flappy Bird, despite the high-dimensional sensory input. The agent is not given information about what the bird or pipes look like - it must learn these representations and directly use the input and score to develop an optimal strategy. Our agent uses a convolutional neural network to evaluate the Q-function for a variant of Q-learning, and he shows that it is able to achieve super-human performance. Furthermore, he discusses difficulties and potential improvements with deep reinforcement learning.

The related work in this area is primarily by Google Deepmind. Mnih et al. are able to successfully train agents to play the Atari 2600 games using deep reinforcement learning, surpassing human expert-level on multiple games. These works inspired this project, which is heavily modeled after their approach. They use a deep Q-network (DQN) to evaluate the Qfunction for Q-learning and also use experience replay to de-correlate experiences. Their approach is essentially state-of-the-art and was the main catalyst for deep reinforcement learning, after which many papers tried to make improvements. The main strength is that they were able to train an agent despite extremely high dimensional input (pixels) and no specification about intrinsic game parameters. In fact, they are able to outperform a human expert on three out of seven Atari 2600 games. However, further improvements involve prioritizing experience replay, more efficient training, and better stability when training.They also tried to address the stability issues by clipping the loss to +1 or 1, and by updating the target network once in every C updates to the DQN rather than updating the target network every iteration.

# Chapter 3

# System Requirements and Analysis

## 3.1 Software Requirements

### 3.1.1 Operating system

This project is developed in a system running windows 10 operating system. It can run on any other operating system which supports the tools and technologies mentioned in the next section.

### 3.1.2 Python

The implementation is done using python language using anaconda and Sublime text editor. Python is a high-level, interpreted, interactive and object-oriented scripting language. Python is designed to be highly readable. It uses English keywords frequently where as other languages use punctuation, and it has fewer syntactical constructions than other languages.Python features a dynamic type system and automatic memory management. It supports multiple programming paradigms, including object-oriented, imperative, functional and procedural, and has a large and comprehensive standard library.

### 3.1.3 Anaconda

Anaconda is a free and open-source distribution of the Python and R programming languages for scientific computing (data science, machine learning applications, large-scale data processing, predictive analytics, etc.), that aims to simplify package management and deployment. Package versions are

managed by the package management system conda. The Anaconda distribution is used by over 12 million users and includes more than 1400 popular data-science packages suitable for Windows, Linux, and MacOS.

Anaconda is a fairly sophisticated software. It supports installation from local and remote sources such as CDs and DVDs, images stored on a hard drive, NFS, HTTP, and FTP.A variety of advanced storage devices including LVM, RAID, iSCSI, and multipath are supported from the partitioning program. Anaconda provides advanced debugging features such as remote logging, access to the python interactive debugger, and remote saving of exception dumps.

### 3.1.4 Sublime Text Editor

Sublime Text is a proprietary cross-platform source code editor with a Python application programming interface (API). It natively supports many programming languages and markup languages, and functions can be added by users with plugins, typically community-built and maintained under free-software licenses.

The following is a list of features of Sublime Text:

- "Goto Anything," quick navigation to files, symbols, or lines.
- "Command palette" uses adaptive matching for quick keyboard invocation of arbitrary commands.
- Simultaneous editing: simultaneously make the same interactive changes to multiple selected areas.
- Python-based plugin API.
- Project-specific preferences.
- Extensive customizability via JSON settings files, including project-specific and platform-specific settings.
- Cross-platform (Windows, macOS, and Linux) and Supportive Plugins for cross-platform.
- Compatible with many language grammars from TextMate

### 3.1.5 Hardware requirements

The hardware requirements for this project are as follows:

- Processor 64 bit, 4 core, 2.00 GHZ
- RAM 8 GB
- Hard disk 14 GB for installation. No free disk space required for running and production use.
- Input Device: Standard keyboard and mouse
- Output Device: Monitors with decent resolution

# Chapter 4

# Tools and Technology Used

### 4.0.1 TensorFlow

TensorFlow is an open source software library for numerical computation using data flow graphs. Nodes in the graph represent mathematical operations, while the graph edges represent the multidimensional data arrays (tensors) communicated between them. The flexible architecture allows you to deploy computation to one or more CPUs or GPUs in a desktop, server, or mobile device with a single API. It is used for machine learning applications such as neural networks. It is used for both research and production at Google, often replacing its closed-source predecessor, DistBelief.

TensorFlow computations are expressed as stateful dataflow graphs. The name TensorFlow derives from the operations that such neural networks perform on multidimensional data arrays. These arrays are referred to as "tensors".

TensorFlow separates the definition of computations from their execution even further by having them happen in separate places: a graph defines the operations, but the operations only happen within a session. Graphs and sessions are created independently. A graph is like a blueprint, and a session is like a construction site.

How is tensorflow different from Python? Or why use tensorflow in the first place?

To do efficient numerical computing in Python, we typically use libraries like NumPy that do expensive operations such as matrix multiplication outside Python, using highly efficient code implemented in another language. Unfortunately, there can still be a lot of overhead from switching back to Python every operation. This overhead is especially bad if you want to run computations on GPUs or in a distributed manner, where there can be a high cost to transferring data. TensorFlow also does its heavy lifting outside

Python, but it takes things a step further to avoid this overhead. Instead of running a single expensive operation independently from Python, TensorFlow lets us describe a graph of interacting operations that run entirely outside Python. This approach is similar to that used in Theano or Torch.

Another exciting feature of tensorflow is a TensorBoard which helps to create diagrammatic explanation. TensorBoard reads the name field that is stored inside each operation (quite distinct from Python variable names). These TensorFlow names can be used and switched to more conventional Python variable names. TensorBoard works by looking at a directory of output created from TensorFlow sessions. TensorBoard runs as a local web app, on port 6006. (6006 is goog upside-down.) If you go in a browser to localhost:6006/#graphs you should see a diagram of the graph you created in TensorFlow.

### 4.0.2 Open CV

OpenCV (Open source computer vision) is a library of programming functions mainly aimed at real-time computer vision. Originally developed by Intel, it was later supported by Willow Garage then Itseez (which was later acquired by Intel). The library is cross-platform and free for use under the open-source BSD license. OpenCV supports the deep learning frameworks TensorFlow, Torch/PyTorch and Caffe.

OpenCV runs on the following desktop operating systems: Windows, Linux, macOS, FreeBSD, NetBSD, OpenBSD. OpenCV runs on the following mobile operating systems: Android, iOS, Maemo, BlackBerry 10. The user can get official releases from SourceForge or take the latest sources from GitHub. OpenCV uses CMake. If the library finds Intel's Integrated Performance Primitives on the system, it will use these proprietary optimized routines to accelerate itself. A CUDA-based GPU interface has been in progress since September 2010. An OpenCL-based GPU interface has been in progress since October 2012, documentation for version 2.4.13.3 can be found at docs.opencv.org.

OpenCV is written in C++ and its primary interface is in C++, but it still retains a less comprehensive though extensive older C interface. There are bindings in Python, Java and MATLAB/OCTAVE. The API for these interfaces can be found in the online documentation. Wrappers in other languages such as C#, Perl, Ch, Haskell, and Ruby have been developed to encourage adoption by a wider audience. Since version 3.4, OpenCV.js is a JavaScript binding for selected subset of OpenCV functions for the web platform. All of the new developments and algorithms in OpenCV are now developed in the C++ interface.

Compared to other languages like C/C++, Python is slower. But another important feature of Python is that it can be easily extended with C/C++. This feature helps us to write computationally intensive codes in C/C++ and create a Python wrapper for it so that we can use these wrappers as Python modules. This gives us two advantages: first, our code is as fast as original C/C++ code (since it is the actual C++ code working in background) and second, it is very easy to code in Python. This is how OpenCV-Python works, it is a Python wrapper around original C++ implementation.

And the support of Numpy makes the task more easier. Numpy is a highly optimized library for numerical operations. It gives a MATLAB-style syntax. All the OpenCV array structures are converted to-and-from Numpy arrays. So whatever operations you can do in Numpy, you can combine it with OpenCV, which increases number of weapons in your arsenal. Besides that, several other libraries like SciPy, Matplotlib which supports Numpy can be used with this. So OpenCV-Python is an appropriate tool for fast prototyping of computer vision problems.

### 4.0.3 Pygame

Pygame is a cross-platform set of Python modules designed for writing video games. It includes computer graphics and sound libraries designed to be used with the Python programming language.

Pygame uses the Simple DirectMedia Layer (SDL) library, with the intention of allowing real-time computer game development without the low-level mechanics of the C programming language and its derivatives. This is based on the assumption that the most expensive functions inside games, can be abstracted from the game logic, making it possible to use a high-level programming language, such as Python, to structure the game.

Other features that SDL doesn't have include vector math, collision detection, 2d sprite scene graph management, MIDI support, camera, pixel array manipulation, transformations, filtering, advanced freetype font support, and drawing. Applications using pygame can run on Android phones and tablets with the use of Pygame Subset for Android (pgs4a). Sound, vibration, keyboard, and accelerometer are supported on Android.

# Chapter 5

# System Design

## 5.1 Deep Q-Network

Deep Q-Network 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. Neural networks are exceptionally good at coming up with good features for highly structured data. The project could be represented with our Q-function with a neural network, that takes the state (four game screens) and action as input and outputs the corresponding Q-value. Alternatively we could take only game screens as input and output the Q-value for each possible action. This approach has the advantage, that if we want to perform a Q-value update or pick the action with the highest Q-value, we only have to do one forward pass through the network and have all Q-values for all actions available immediately[16]. The same is represented in figure 5.1 as follows.

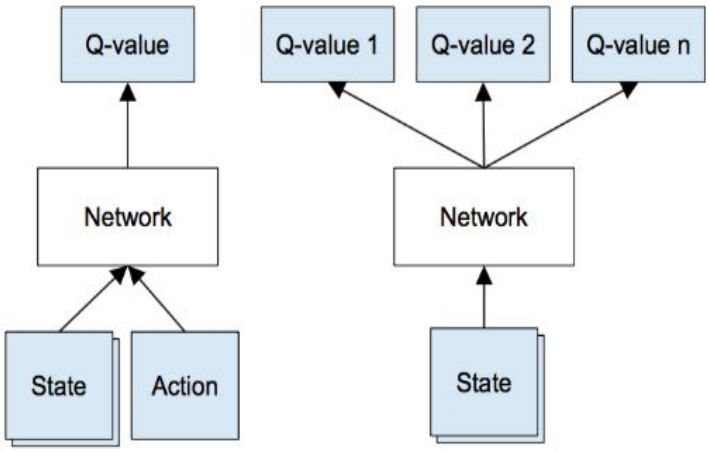


Figure 5.1: Left:Naive formulation of deep Q-network. Right:More optimized architecture of deep Q-network

## 5.2 Deep Q-Network Algorithm

The pseudo-code for the Deep Q Learning algorithm, can be found below:

```
Initialize replay memory D to size N
Initialize action-value function Q with random weights
for episode = 1, M do
    Initialize state s_1
    for t = 1, T do
        With probability ϵ select random action a_t
        otherwise select a_t=max_a  Q(s_t,a; θ_i)
        Execute action a_t in emulator and observe r_t and s_(t+1)
        Store transition (s_t,a_t,r_t,s_(t+1)) in D
        Sample a minibatch of transitions (s_j,a_j,r_j,s_(j+1)) from D
        Set y_j:=
            r_j for terminal s_(j+1)
            r_j+γ*max_(a^' )  Q(s_(j+1),a'; θ_i) for non-terminal s_(j+1)
        Perform a gradient step on (y_j-Q(s_j,a_j; θ_i))^2 with respect to θ
    end for
end for
```

## 5.3 Experiments

### 5.3.1 Environment

Since deep Q-network is trained on the raw pixel values observed from the game screen at each time step, [3] finds that remove the background appeared in the original game can make it converge faster. This process can be visualized as shown in the figure 5.2

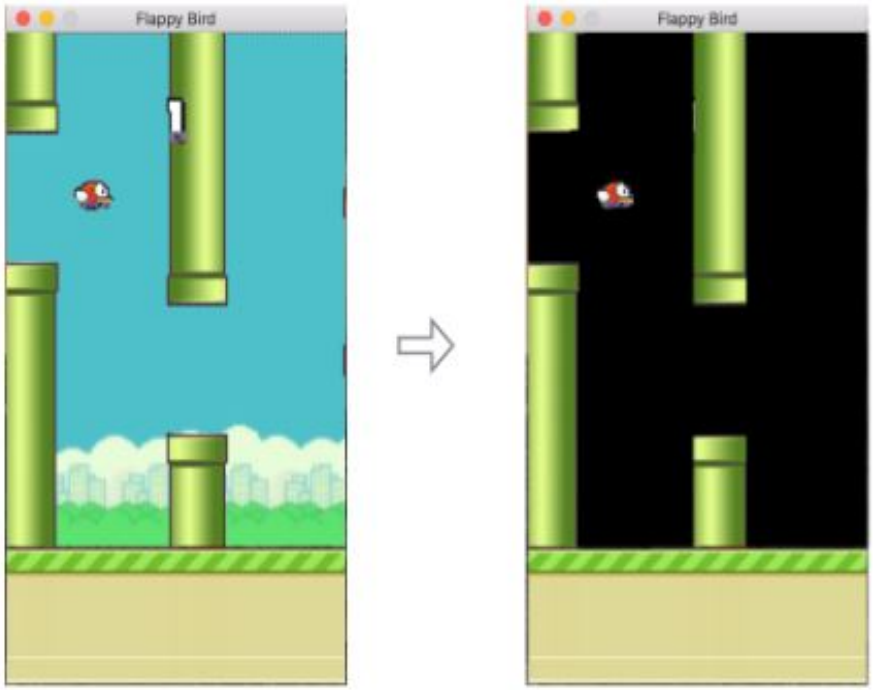


Figure 5.2: Environment visualization

### 5.3.2 Network Architecture

According to the proposed solution, we first need to preprocess the game screens with following steps:

- Convert image to grayscale
- Resize image to 80x80
- Stack last 4 frames to produce an 80x80x4 input array for network

The architecture of the network is shown in the figure 5.3. The first layer convolves the input image with an 8x8x4x32 kernel at a stride size of 4. The output is then put through a 2x2 max pooling layer. The second layer convolves with a 4x4x32x64 kernel at a stride of 2. Then further the max pooling is done again. The third layer convolves with a 3x3x64x64 kernel at a stride of 1. Then it has to be max pooled one more time. The last hidden layer consists of 256 fully connected ReLU nodes.

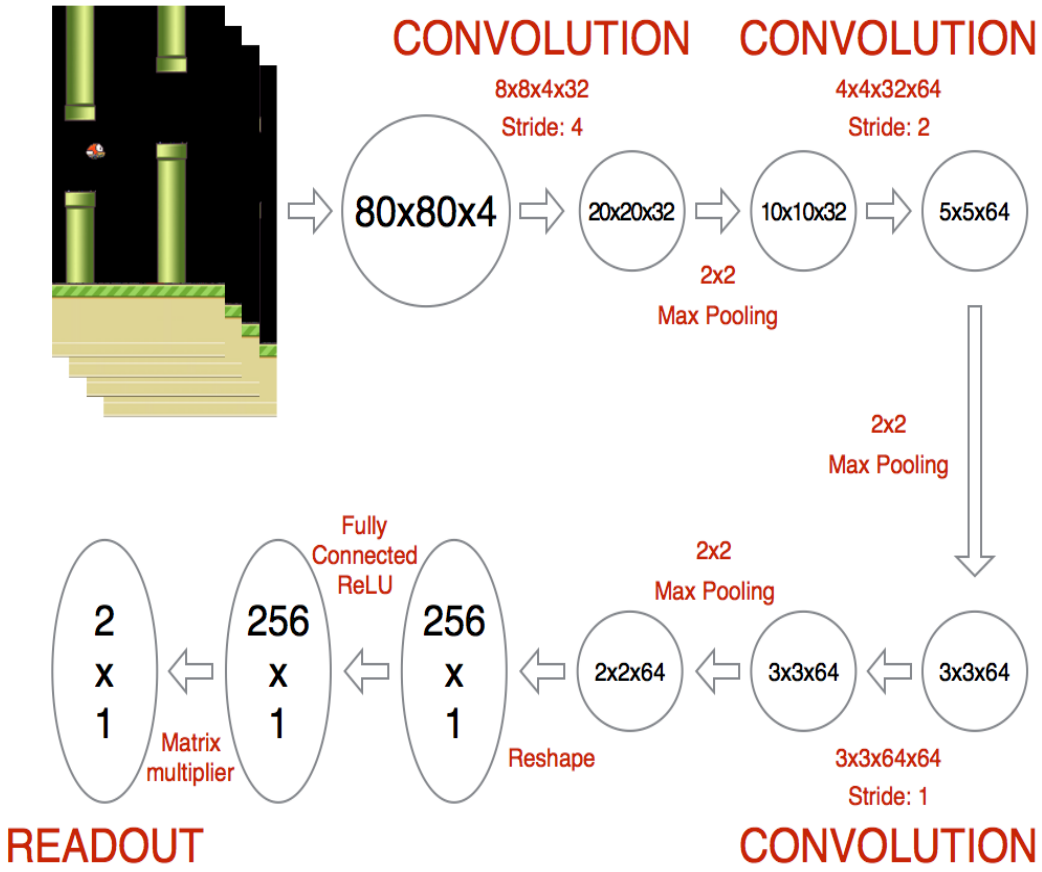


Figure 5.3: Architecture of the network

The final output layer has the same dimensionality as the number of valid actions which can be performed in the game, where the 0th index always corresponds to doing nothing. The values at this output layer represent the Q function given the input state for each valid action. At each time step, the network performs whichever action corresponds to the highest Q value using an epsilion greedy policy.

### 5.3.3 Training

At first, we need to initialize all weight matrices randomly using a normal distribution with a standard deviation of 0.01, then set the replay memory with a max size of 5,00,00 experiences.

Then we must start training by choosing actions uniformly at random for the first 10,000 time steps, without updating the network weights. This allows the system to populate the replay memory before training begins.

Here we also must note that unlike the other models, which initialize epsilion equals to 1, we linearly anneal epsilion from 0.1 to 0.0001 over the course of the next 30,00,000 frames. The reason why we set it this way is that the agent can choose an action every 0.03s (FPS=30) in our game, high epsilion will make it flap too much and thus keeps itself at the top of the game screen and finally bump the pipe in a clumsy way. This condition will make Q function converge relatively slow since it only start to look other conditions when epsilion is low. However, in other games, initializing epsilion to 1 is more reasonable.

During training time, at each time step, the network samples minibatches of size 32 from the replay memory to train on, and performs a gradient step on the loss function described above using the Adam optimization algorithm with a learning rate of 0.000001. After annealing finishes, the network continues to train indefinitely, with epsilion fixed at 0.001[4].

# Chapter 6

# System Implementation

## 6.1 Asynchronous n-step Q-learning

Pseudocode for our variant of multi-step Q-learning is shown in Supplementary Algorithm S2. The algorithm is somewhat unusual because it operates in the forward view by explicitly computing nstep returns, as opposed to the more common backward view used by techniques like eligibility traces . It is found that using the forward view is easier when training neural networks with momentum-based methods and backpropagation through time. In order to compute a single update, the algorithm first selects actions using its exploration policy for up to tmax steps or until a terminal state is reached. This process results in the agent receiving up to tmax rewards from the environment since its last update.

The algorithm then computes gradients for n-step Q-learning updates for each of the state-action pairs encountered since the last update. Each n-step update uses the longest possible n-step return resulting in a one-step update for the last state, a two-step update for the second last state, and so on for a total of up to tmax updates. The accumulated updates are applied in a single gradient step.

## 6.2 Algorithm

```
    ε: 0<ε<1, Percent of time to use exploration
    S, A: Sets of states and actions
    α: Learning rate
    N: Number of steps
    Tmax: Number of total steps  ### Output:

    Q function
T <-- 0
Repeat:
    t <-- 0
    tstart <-- t
    previous_rewards <-- []
    previous_states <-- []
    previous_actions <-- []
    Get state s
    Repeat:
        Choose random e, 0 < e < 1
        If e < ε
            a <-- random.choose(As)
        Else
            a <-- argmaxQ(s,a)

        Take action a and observe r and s'
```

Figure 6.1: Supplementary Algorithm S2

```
        previous_rewards[i] <-- r
        previous_states[i] <-- s
        previous_actions[i] <-- a
        t <-- t+1
        T <-- T+1
        s <-- s'
    Until (s is terminal) or (t-tstart==N)

    If s is terminal
        R <-- 0
    Else
        R <-- max(Q(s,a))

    For i = t-1 : tstart
        r_i <--  previous_rewards[i]
        s_i <--  previous_states[i]
        a_i <--  previous_actions[i]
        R <-- r_i + γR
        # Update Q values
        Q(s_i,a_i) <-- Q(s_i,a_i) - α(Q(s_i,a_i) - R)

Until T>Tmax
Return Q
```

Figure 6.2: Supplementary Algorithm S2 (cont..)

# Chapter 7

# System Testing and Results Analysis

The project would present asynchronous versions of reinforcement learning algorithm and show that they are able to train neural network controllers on a variety of domains in a stable manner. Our results would show that in our proposed framework stable training of neural networks through reinforcement learning is possible with both value-based and policy-based methods, off-policy as well as onpolicy methods and in discrete as well as continuous domains.

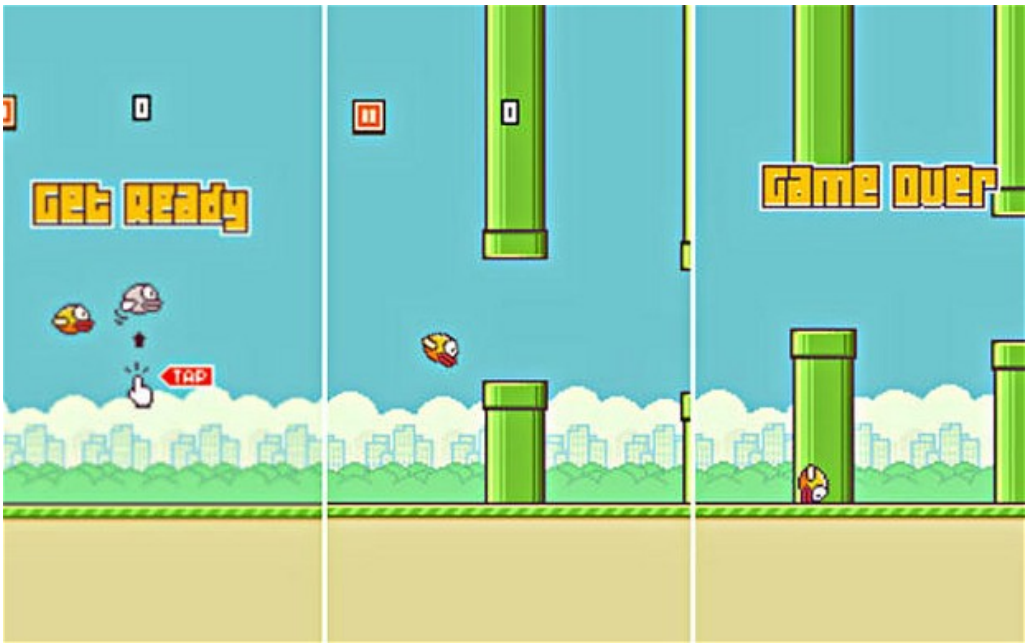


Figure 7.1: Flappy Bird game

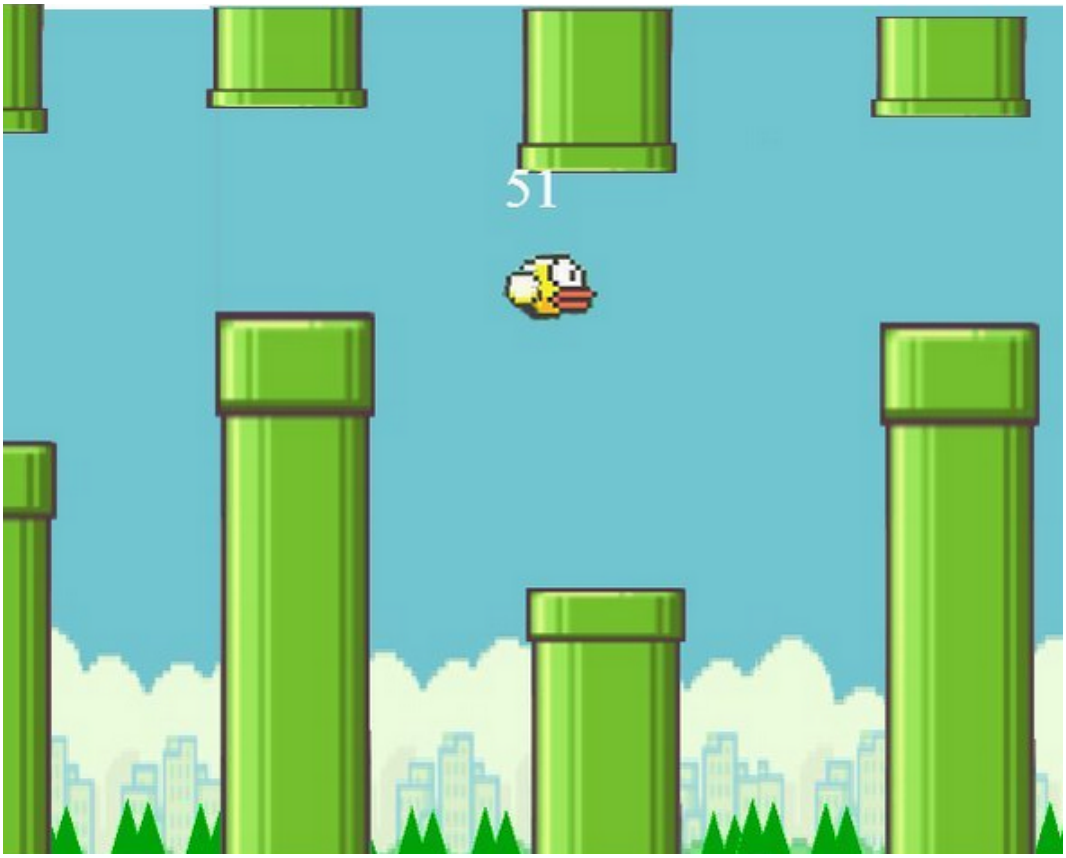


Figure 7.2: Different stages in the game

## 7.1 Robustness and Stability

Finally, we analyzed the stability and robustness of the four proposed asynchronous algorithms. For each of the four algorithms we trained models on games of different domains using 50 different learning rates and random initializations. Figure 7.3 shows plots for these methods. There is usually a range of learning rates for each method and game combination that leads to good scores, indicating that all methods are quite robust to the choice of learning rate and random initialization. The fact that there are virtually no points with scores of 0 in regions with good learning rates indicates that the methods are stable and do not collapse or diverge once they are learning.

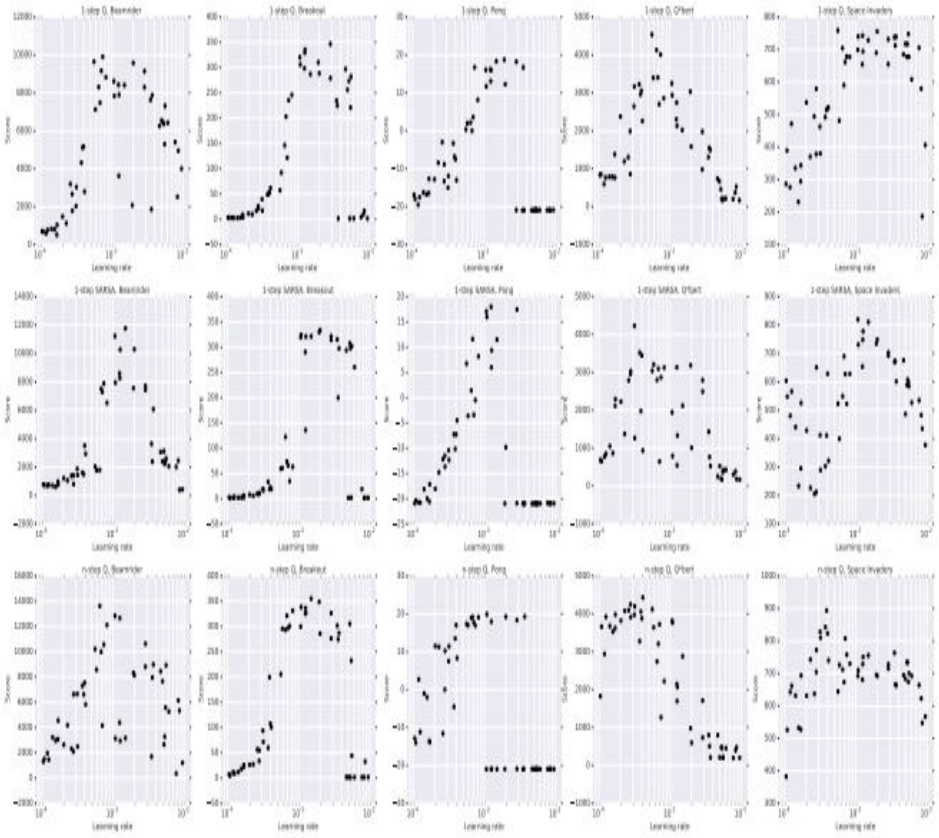

Figure 7.3: Scatter plots of scores obtained by one-step Q, one-step Sarsa, and n-step Q learning

# Chapter 8

# Conclusion and future work

The project has presented asynchronous versions of three standard reinforcement learning algorithms and also showed that they are able to train neural network controllers on a variety of domains in a stable manner. Our proposed framework provides stable training of neural networks through reinforcement learning and it also performs with both valuebased and policy-based methods, off-policy as well as onpolicy methods, and in discrete as well as continuous domains. Our future work would be to train on the other diffrent Atari domains as well using 16 CPU cores and to show that the proposed asynchronous algorithms train faster than DQN trained on an Nvidia K40 GPU, with A3C surpassing the current state-of-the-art in half the training time.

One of our main findings is that using parallel actorlearners to update a shared model had a stabilizing effect on the learning process of the three value-based methods we considered. While this shows that stable online Q-learning is possible without experience replay, which was used for this purpose in DQN, it does not mean that experience replay is not useful. Incorporating experience replay into the asynchronous reinforcement learning framework could substantially improve the data efficiency of these methods by reusing old data. This could in turn lead to much faster training times in domains like TORCS where interacting with the environment is more expensive than updating the model for the architecture we used.

Combining other existing reinforcement learning methods or recent advances in deep reinforcement learning with our asynchronous framework presents many possibilities for immediate improvements to the methods we presented. While our n-step methods operate in the forward view (Sutton and Barto, 1998)[12] by using corrected n-step returns directly as targets, it has been more common to use the backward view to implicitly combine different returns through eligibility traces (Watkins, 1989; Sutton

and Barto,1998; Peng and Williams, 1996)[8]. The asynchronous advantage actor-critic method could be potentially improved by using other ways of estimating the advantage function, such as generalized advantage estimation of (Schulman et al., 2015b). All of the value-based methods we investigated could benefit from different ways of reducing overestimation bias of Q-values (Van Hasselt et al., 2015; Bellemare et al., 2016). Yet another, more speculative, direction is to try and combine the recent work on true online temporal difference methods (van Seijen et al., 2015) with nonlinear function approximation[18].

In addition to these algorithmic improvements, a number of complementary improvements to the neural network architecture are possible. The dueling architecture of (Wang et al., 2015)[17] has been shown to produce more accurate estimates of Q-values by including separate streams for the state value and advantage in the network. The spatial softmax proposed by (Levine et al., 2015)[6] could improve both value-based and policy-based methods by making it easier for the network to represent feature coordinates.

# Chapter 9

# Appendix

## 9.1 Appendix A: Project Team Details

| Project Title | Asynchronous method for deep reinforcement learning | | |
|---|---|---|---|
| USN | Team Members Name and CGPA | E-Mail Id | Mobile Number |
| 4JC15CS068 | Nithin Prabhu G (9.08) | nithingprabhu@gmail.com | 9886722133 |
| 4JC16CS419 | Shwetha U (7.14) | shwethanaik22@gmail.com | 9632620368 |
| 4JC16CS422 | Yathish G (7.3) | Yathishg2016@gmail.com | 7996305076 |

Figure 9.1: Team Photograph

## 9.2 Appendix B: COs, POs and PSOs Mapping for the Project Work (CS84P)

### 9.2.1 Course Outcomes:

**CO1:** Formulate the problem definition, conduct literature review and apply requirements analysis.
**CO2:** Develop and implement algorithms for solving the problem formulated.
**CO3:** Comprehend, present and defend the results of exhaustive testing and explain the major findings.

### 9.2.2 Program Outcomes:

**PO1:** Apply knowledge of computing, mathematics, science, and foundational engineering concepts to solve the computer engineering problems.
**PO2:** Identify, formulate and analyze complex engineering problems.
**PO3:** Plan, implement and evaluate a computer-based system to meet desired societal needs such as economic, environmental, political, healthcare and safety within realistic constraints.
**PO4:** Incorporate research methods to design and conduct experiments to investigate real-time problems, to analyze, interpret and provide feasible conclusion.
**PO5:** Propose innovative ideas and solutions using modern tools.
**PO6:** Apply computing knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to professional engineering practice.
**PO7:** Analyze the local and global impact of computing on individuals and organizations for sustainable development.
**PO8:** Adopt ethical principles and uphold the responsibilities and norms of computer engineering practice.
**PO9:** Work effectively as an individual and as a member or leader in diverse teams and in multidisciplinary domains.
**PO10:** Effectively communicate and comprehend.
**PO11:** Demonstrate and apply engineering knowledge and management principles to manage projects in multidisciplinary environments.
**PO12:** Recognize contemporary issues and adapt to technological changes for lifelong learning.

### 9.2.3 Program Specific Outcomes:

**PSO1: Problem Solving Skills:** Ability to apply standard practices and mathematical methodologies to solve computational tasks, model real world problems in the areas of database systems, system software, web technologies and Networking solutions with an appropriate knowledge of Data structures and Algorithms.
**PSO2: Knowledge of Computer Systems:** An understanding of the structure and working of the computer systems with performance study of various computing architectures.
**PSO3: Successful Career and Entrepreneurship:** The ability to get acquaintance with the state of the art software technologies leading to entrepreneurship and higher studies.
**PSO4: Computing and Research Ability:** Ability to use knowledge in various domains to identify research gaps and to provide solution to new ideas leading to innovations.

### 9.2.4 Justification for the mapping

The first CO is related to problem definition, literature survey and requirement analysis. Planning the project such that it meets the needs of society, by considering all the constraints is very relevant for this. Investigating real time problems and incorporating our findings in literature survey plays an important role as well. Understanding the constraints of the environment in which the system will be used plays a crucial role in deciding the requirements and using latest technology to make our implementation better is of high relevance.

The second CO, design and implementation highly depends on the way we apply already acquired knowledge about computing, mathematics, etc., the innovation we bring in to our implementation, make our implementation adaptable to technological changes that might happen in future and the way we look at the problem and apply our knowledge. The ability to analyze global and local impact of the system, ability to uphold the ethical principles of engineering practices, the way in which we communicate and comprehend the concepts, the way in which the project is handled in multidisciplinary environments hold great relevance in defending our work and explaining major findings during our project.

Note:

1. Scale 1 Low relevance
2. Scale 2 Medium relevance
3. Scale 3 High relevance

| SEM | SUBJECT | CODE | CO | PO1 | PO2 | PO3 | PO4 | PO5 | PO6 | PO7 | PO8 | PO9 | PO10 | PO11 | PO12 | PSO1 | PSO2 | PSO3 | PSO4 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| VIII | Project Work | CS84P | CO1 | 1 | 3 | 3 | 1 | 3 | 2 | 2 | 2 | 3 | 1 | 1 | 1 | 2 | 3 | 1 | 1 |
| | | | CO2 | 3 | 3 | 1 | 3 | 1 | 2 | 1 | 1 | 3 | 1 | 1 | 1 | 3 | 2 | 1 | 1 |
| | | | CO3 | 2 | 1 | 3 | 1 | 1 | 2 | 3 | 3 | 3 | 3 | 2 | 3 | 1 | 1 | 3 | 3 |

The following subjects helped us in developing our project:

- To formulate the problem definition, conduct literature review and apply requirements analysis that we gained the knowledge from Software Engineering and System Software.
- To Develop and implement algorithms for solving the problem formulated , we gained the knowledge from subjects such as Neural Networks, Object Oriented Programming, Machine Learning, Data Warehousing and Data Mining.
- To Comprehend, present and defend the results of exhaustive testing and explain the major findings, we gained the knowledge from Engineering Mathematics, Web Technologies and Operating Systems.

# Chapter 10

# References


1. Bellemare, Marc G, Naddaf, Yavar, Veness, Joel, and Bowling, Michael. The arcade learning environment:An evaluation platform for general agents. Journal of Artificial Intelligence Research, 2012.

2. Chavez, Kevin, Ong, Hao Yi, and Hong, Augustus. Distributed deep q-learning. Technical report, Stanford University, June 2015.

3. Degris, Thomas, Pilarski, Patrick M, and Sutton, Richard S. Model-free reinforcement learning with continuous action in practice. In American Control Conference (ACC), 2012, pp. 21772182. IEEE, 2012.

4. Kevin Chen. Deep Reinforcement Learning for Flappy Bird

5. Koutnk, Jan, Schmidhuber, Jrgen, and Gomez, Faustino. Evolving deep unsupervised convolutional networks for vision-based reinforcement learning. In Proceedings of the 2014 conference on Genetic and evolutionary computation, pp. 541548. ACM, 2014.

6. Levine, Sergey, Finn, Chelsea, Darrell, Trevor, and Abbeel, Pieter. End-to-end training of deep visuomotor policies. arXiv preprint arXiv:1504.00702, 2015.

7. Mnih Volodymyr, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level Control through Deep Reinforcement Learning. Nature, 529-33, 2015

8. Peng, Jing and Williams, Ronald J. Incremental multi-step q-learning. Machine Learning, 22(1-3):283290, 1996.

9. Recht, Benjamin, Re, Christopher, Wright, Stephen, and Niu, Feng. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In Advances in Neural Information Processing Systems, pp. 693701, 2011.

10. Rummery, Gavin A and Niranjan, Mahesan. On-line qlearning using connectionist systems. 1994.

11. Schaul, Tom, Quan, John, Antonoglou, Ioannis, and Silver, David. Prioritized experience replay. arXiv preprint arXiv:1511.05952, 2015.

12. Sutton, R. and Barto, A. Reinforcement Learning: an Introduction. MIT Press, 1998.

13. Todorov, E. MuJoCo: Modeling, Simulation and Visualization of Multi-Joint Dynamics with Contact (ed 1.0). Roboti Publishing, 2015.

14. Tomassini, Marco. Parallel and distributed evolutionary algorithms: A review. Technical report, 1999.

15. Van Hasselt, Hado, Guez, Arthur, and Silver, David. Deep reinforcement learning with double q-learning. ArXiv preprint arXiv:1509.06461, 2015.

16. Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing Atari with Deep Reinforcement Learning. NIPS, Deep Learning workshop

17. Wang, Z., de Freitas, N., and Lanctot, M. Dueling Network Architectures for Deep Reinforcement Learning. ArXiv e-prints, November 2015.

18. Williams, Ronald J and Peng, Jing. Function optimization using connectionist reinforcement learning algorithms. Connection Science, 1991