Technische Universiteit
Eindhoven
University of Technology

Thesis Report

Design, implementation, optimization, and evaluation of energy-efficient AI controllers and testing their feasibility on a real continuous-time dynamical system.

Author:
Yash Kamate (ID: 1353578)

Supervisors:
Dr. Dip Goswami (TU/e)
Dr. Sander Bohte (CWI)

Department of Mathematics and Computer Science
Masters in Embedded Systems

Eindhoven, Tuesday 11th August, 2020

Acknowledgement

This thesis is an outcome of a year of hard work, dedication, and persistence. It gives me immense pleasure to take this opportunity to show my gratitude to everyone who has helped me achieve this success.

I would firstly like to thank my university supervisor, Dr. Dip Goswami, for allowing me to take up this opportunity and present my contribution to the field of AI and control systems. His encouragement throughout the thesis helped me in building up my confidence. My deepest thanks also go to Dr. Henk Corporaal who believed in my capabilities and showed trust in me to complete the research.

I must express my very profound gratitude to my company supervisor Dr. Sander Bohte, who carried me through all the difficult situations in my research and helped me with the right skills to conduct the study. His continuous feedbacks and regular meetings have helped me correctly steer my research.

I wish to extend my special thanks to my mentor Bojian Yin who has been supporting me daily throughout the thesis. I especially thank him for being patient in answering all my questions and helping me explore new approaches. The numerous brainstorming sessions that we had were vital in inspiring me to think outside the box. Without his passionate involvement and support, this research could not have been successfully completed.

The situation had been tough due to the pandemic and working from home with the hardware was indeed tricky. I wish to show my appreciation to all those people who did not refrain from helping me in the areas where I lacked expertise. I would like to thank Sayandip De and Kanishkan Vadival for their precious suggestions on energy profiling tools. Furthermore, I thank Sajid Mohamed from the bottom of my heart for helping me during the initial phase of the thesis. Without his valuable guidance in laying out the foundation for the research, it would have been difficult to kick off the study.

Last but not the least, I thank my family for being patient at times when I was stressed. Their never-ending encouragement, support, and attention played a key role in the completion of my thesis. Finally, I cannot begin to express my gratitude towards my friends (also from India) who have brought great care, support, and room for enjoyment during this very intensive yet fruitful academic year.

Abstract

With an astonishing success of AI-based controllers on simulated dynamical systems, very little has been explored in their applicability to real-world continuous-time dynamical control systems. This study investigates the practicality of two popular categories of AI controllers namely Deep Neural Networks (DNNs) and Spiking Neural Networks (SNNs) on a physical continuous-time dynamical system with real-time constraints. The goal of the thesis is achieved through the design, implementation, optimization, and evaluation of the energy efficiency of the mentioned category of AI controllers. Our research goal includes obtaining minimal DNN and SNN controllers optimized in terms of network size and connectivity to control the physical system. The research further extends to striking comparison in terms of energy consumption of DNN and SNN controllers when they execute control over the same dynamical system and at the same performance.

Initially, we designed and built a real cartpole system which was the target system to be controlled by the AI controllers. We studied, designed, and implemented two Reinforcement Learning (RL) algorithms namely Deep Q-Network (DQN) and Neural Fitted-Q iteration (NFQ) on the cartpole system. We also implemented an Evolutionary algorithm, the Genetic Algorithm (GA) to produce AI controllers. This algorithm was chosen to produce the DNN and SNN controllers as it provided the best results compared to the former two RL algorithms. The designed AI controllers were optimized through pruning and the most optimal controllers were selected to calculate the energy consumption. We defined two energy models based on which the DNN and SNN controllers were profiled. Finally, the AI controllers were compared based on the energy consumptions and the results are discussed.

We obtained a model-based DQN controller that demonstrated an average balance time of 2.2 seconds, a model-free NFQ controller that also showed an average balance time of 2.2 seconds, and a GA controller that exhibited an average balance time of 2.42 seconds on the real cartpole system. We produced a DNN controller using GA that could perform an average balance time of 2.42 seconds whereas an SNN controller using GA that could balance the pole on the real cartpole system for an average of 2.48 seconds. The minimal size of both the controllers obtained that was able to balance the cartpole system was 10 neurons with 4 neurons in the input layer, 4 neurons in the hidden layer, and 2 neurons in the output layer. These AI controllers were profiled for energy consumption based on the number of MAC operations and internal calculations including memory loads and stores. Our analysis showed that the DNN controller consumed 1.5 times the energy as that of the SNN controller in terms of energy required for the computations and memory accesses. However, in the case of the number of MAC operations, the SNN controller was efficient as it took 0.45 times the number of MAC operations as that of DNN.

The primary limitation of the thesis is the use of comparatively simpler hardware to design a cartpole system. The cartpole system was designed using a LEGO Education Kit with the AI controllers running on a non-real-time embedded Linux OS. Simply put, the AI controllers are running on top of a non-real-time hardware setup to control a real-time problem which resulted in a noticeably lower benchmark of performance set by the AI controllers.

Our research concludes that the AI controllers are capable of performing well on real-time tasks

in the real-world provided that the hardware of the control system has a real-time nature. We could judge the practicality of AI controllers on practical dynamical control systems by implementing two AI controllers belonging to the category of DNN and SNN. We conclude that using SNN and DNN controllers on general-purpose processors (i.e. non-neuromorphic processors) results in proving DNN controllers as energy-efficient solutions than SNN controllers in terms of core computations and memory load and store operations. On the contrary, SNN controllers show great efficiency in terms of the number of MAC operations that are the most power-consuming operations on a general 45 nm process.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Deep Reinforcement Learning (DRL) has shown great advancement in the ability to control complex systems. Commonly, Deep Neural Networks (DNNs) are used to compute the controller actions in DRL. However, most DNNs establish control over the system in a discrete fashion. The state of the system to be controlled is sampled at every sampling period and fed to the DNN controller. The DNN controller uses the state input and calculates an action from a discrete set of actions to be given to the system such that the system can be controlled within the predefined limits. Precisely, at every sampling period, the neural network passes the input values from the input layer (i.e. 1^{st} layer) to the consequent layer (called the hidden layer(s)) through a sequence of calculations until it reaches the final layer (output layer) to produce a discrete action. As a result, there will be an equal number of state samples and actions generated in the case of a DNN controller.

Another class of Artificial Neural Networks (ANNs) called Spiking Neural Networks (SNNs) exist which offer continuous control over the system. However, the term 'continuous control' holds a different meaning in this aspect. Similar to a DNN, an SNN is also fed with a state sample taken at every sampling time. The number of discrete actions produced by the SNN controller is also equal to the number of state samples. The major difference lies in the manner in which the SNNs calculate the control action. Simply put, the frequency with which switching of the actions occurs between two consecutive sampling periods in SNNs is less as compared to DNNs. This quality of SNNs is attributed to the sparsity of its neuron firing. The less frequent switching of actions brings about the effect of a continuous control over the system such that the controller acts on the system only when it is required to.

The ability of the mentioned controllers in controlling a dynamical system using different methodologies gives rise to the need for performance evaluation of each of these controllers on a control application. We aim to compare the energy efficiencies of SNN and DNN controllers when they execute control on a target system of a comparable level. We present research on how efficiently can the SNNs be designed with respect to an optimized DNN on a physical dynamical system with low dimensional sensory input. Further, we implement both the controllers on a practical dynamical system to check their feasibility in real-world scenarios. The thesis also aims at minimizing the energy consumptions of both the approaches by optimizing the models through the frameworks used to design them.

1.2 Structure of Thesis

The thesis is organized into 10 chapters and the topics covered in each chapter are as follows.

Chapter 2 formally states the objective or the problem statement of the thesis and enlists the research questions solved by the thesis. The chapter also explains the technical milestones that were set beforehand for a timely completion and tracking the progress of the thesis. In chapter 3, we explain the various concepts and theories concerned to the thesis to build a proper orientation of the reader. We also cover the background and explain the algorithms used in the thesis in detail. Chapter 4 describes the evaluation framework used for the thesis with its hardware and software development steps. The chapter also provides the reasons for the choice of the hardware and enlists the specifications of the designed system.

Chapter 5 entails the implementation procedure of DQN algorithm providing its motivation, experiment results and conclusion. Chapter 6, describes the methodology of implementing NFQ algorithm on the hardware along with its results. On the similar lines, chapter 7 and 8 narrate the implementation procedures of GA and SNN algorithms respectively providing their motivation, experiments and outcomes.

Chapter 9 focuses on optimizing the best controllers obtained from chapters 5, 6, 7, and 8 and describes the attempts undertaken to make the controllers energy efficient. The thesis concludes with evaluating the best DNN and SNN controllers and comparing their performances on the target hardware in the chapter 10. It also describes the energy models that were built to evaluate the energy efficiency of the chosen controllers. A brief discussion on the results and their interpretation is done in the end.

Chapter 2

Problem Statement

2.1 Objective

The thesis aims to check the feasibility of the two state-of-the-art AI controllers namely: Spiking neural network and Deep neural network on a practical control system with real-time constraints. Moreover, it also focuses on establishing an energy comparison of the aforementioned controllers in controlling the dynamical system. Formally stating, the objective of the thesis is,

"Design, implementation, optimization, and evaluation of energy-efficient AI controllers and testing their feasibility on a real continuous-time dynamical system."

- AI controllers: We specifically implement the two categories of AI controllers namely Deep Neural Network (DNN) and Spiking Neural Network (SNN).
- Target platform: The chosen pair of controllers will be implemented on a real-life cartpole system which represents a real-time dynamical system.

2.2 Research Questions

Following are the set of research questions that address the need for research on efficient control on real dynamical systems using AI controllers:

1. Given the current state of the art algorithms, what is the minimal optimized Deep Neural Network (DNN) for robust control of a real Cartpole system where the DNN is optimized in terms of network size and connectivity i.e. weights and biases, with standard energy optimization techniques to achieve a given level of performance?
2. What is the energy consumption of such a network in terms of an energy model that is defined by the number of operations such as multiplication and additions, and number of load and store operations?
3. How do Spiking Neural Networks (SNNs) compare to DNNs in terms of energy consumption as defined by the energy model in question 2, when controlling the same cartpole system at a comparable performance?

2.3 Technical Milestones

To achieve the objective of the thesis, we have to complete several tasks in the given time. In this section, we enlist the milestones that were set to keep a track of the progress and get an idea of possible roadblocks that may arrive during the course of the thesis. Besides, the milestones aid

in providing a proper sequence and they classify the project into different phases. Multiple set of tasks were required to be completed to reach a milestone and keeping that in mind, we outlined them as follows:

1. Overview on the state of the art DNNs used for control applications.
    - Choose a DNN algorithm suitable for the cartpole-control application
2. Check options and feasibility for buying or making a real single-pole cartpole system
    - Build/buy a real-life cartpole system.
    - Adapt the cartpole system to make it compatible with a reinforcement learning agent/algorithm.
3. Design and implementation of a Deep Neural Network on the hardware.
    - Design the chosen Deep Neural Network algorithm.
    - Implement the controller on the simulations.
    - Implement the controller on the hardware.
4. Optimizing the chosen DNN
    - Optimize network structure through pruning and reducing intermediate layers.
    - Quantize the weights and biases.
5. Test and evaluate the AI controllers on the real cartpole model.
    - Make a simple energy model for the control based on
        - Number of operations (additions, multiplications, etc).
        - Number of external and internal memory loads and stores.
    - Profile the controller and determine energy usage.
    - Compare energy and quality to a Spiking Neural Network controller for the cartpole balance problem.

The thesis intends to design the SNN and the DNN controllers, optimize them and use the most optimum versions of both the controllers. However, while doing so, maintaining impartiality by the designer is important and can be a debatable concern at the end of comparison results. To avoid being tendentious towards SNNs or DNNs, the SNN controller will be studied and designed by another student. The designing, implementation, and optimization of the DNN would be carried out by myself. On the completion of the design of the SNN controller, it will also be implemented and optimized by me. Finally, the designs of controllers using both the approaches (SNN and DNN) will be compared and tested by me.

Chapter 3

Related Work

In this chapter, we dive into the concepts of Supervised, Unsupervised, and Reinforcement learning which form the main categories of Machine learning. We explore the relation of Artificial Intelligence and Machine learning and understand the need for Reinforcement learning for our study. Next, we briefly describe the algorithms Deep Q-Learning (DQN), Neural Fitted-Q Iteration (NFQ), and Genetic Algorithm (GA). Finally, we understand the two categories of neural networks: Deep Neural Network (DNN) and Spiking Neural Networks (SNN) which we propose to compare in terms of their performance in this thesis. We end the chapter with the basic concepts of the system under study i.e. the Cartpole System.

3.1 Supervised, Unsupervised, and Reinforcement learning

Information is everywhere. Apart from the human being himself, the machines that are produced, the IoT devices, the social media, the business transactions, and nearly everything that can be supervised produces data in different formats and sizes. This data is collectively termed as 'Big data'. Back in 2001, Doug Laney articulated the definition of Big Data as being 3-dimensional and is represented using the 3 V's namely Volume, Velocity, and Variety [15]. The magic of Big Data does not reside in its quantity but the insights that it can provide. With a proper motive, the applications of Big Data lay in product optimizations, innovation, cost and time reductions, smarter decisions, and many more.

Computers are sophisticated machines that are capable of solving complex problems through the use of various algorithms. Artificial Intelligence (AI) is a process of inducing intelligence into computers by explicitly programming them. We can consider it as a comprehensive set of 'if-else' rules which the computers follow to sense, reason, act, and adapt to a system. However, these computers have certain constraints such as storage, computational capability, etc. The problem arises when these computers, with limited capacities, are allowed to face the demanding Big Data. There arises a point when our traditional algorithms start to fail and coding an exhaustive set of rules becomes impossible. This is the point where Machine learning comes to the rescue. It is a field that amalgamates Artificial Intelligence, computer programming, and philosophy all in one. It fosters the system (i.e. the computer) with the ability to automatically learn and improve from experience without being explicitly programmed. Machine learning's key concept lies in minimizing to eliminating the human-in-the-loop while performing a task. It focuses on building computer programs that learn by observing the data and make decisions about the data through either following the instructions given by the programmer, observing patterns in the data, or direct experience. Owing to the technique used in learning from the data, Machine Learning is broadly divided into three categories namely, Supervised Learning, Unsupervised Learning, and Reinforcement Learning. A pictorial hierarchy of the Artificial Intelligence fields is as shown in figure 3.1.

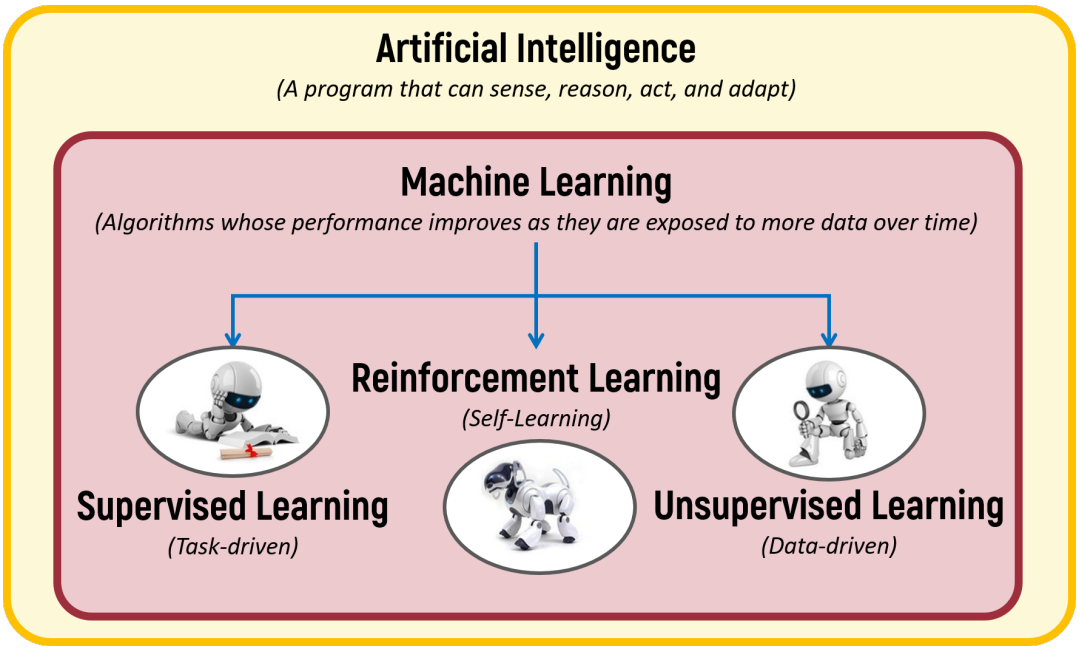


Figure 3.1: Hierarchy of Artificial Intelligence fields. (Graphics Source: [5])

Supervised learning (SL) algorithms rely on the data that comes along with labels. The model that is going to be trained is also termed as the 'agent'. In supervised machine learning, we first provide the model with a subset of raw labeled data called the 'training data set'. The agent is trained to produce a function that would map the inputs to the outputs that correspond to the labels of the input data. Once the agent has learned enough i.e it is able to predict the correct labels of the input data, it is tested on another subset called the 'test data set' to check if it can map the inputs to the outputs correctly. Finally, the knowledge of the agent i.e. the mapping function can be used on unseen future data to predict the labels. Thus, supervised learning is useful when we have a labeled data set.

Unsupervised learning (UL) contrasts supervised learning in the sense that, the data we have is neither classified nor labeled. The idea behind using unsupervised learning is to allow the agent to find hidden patterns in the input data set and try to predict the output. To train the model, we feed it with a training data set without any explicit instructions as to what has to be done with it. Unsupervised learning is used in cases where we expect some relationship between the data in the data set but we cannot directly point it out due to the complexity in the data set. Thus, we ask the model to group the data by finding hidden features within it in a way that makes sense. Therefore, we can choose this type of learning mechanism whenever we have a data set but it is unlabelled and incomplete.

Reinforcement learning (RL) is the category of machine learning that is neither based on supervised nor unsupervised learning. The learning of the agent happens through an 'environment' and improving its 'actions' through the 'rewards' it obtains from the environment. In a reinforcement learning problem, there exists a start-state or an initial state and an end-state or a terminal state. The agent begins in the start state and its task is to reach the end state by providing different actions to the environment. The agent receives appreciation from the environment in terms of rewards if its previous action led it to success or end-state, and does not receive any reward otherwise. Thus, a trial and error approach combined with the delayed rewards guides the agent to automatically determine the ideal behavior to achieve higher rewards. The

agent-environment interaction process in a reinforcement learning problem is as shown in figure 3.2. RL is not completely a supervised learning technique from the fact that it does not rely on a labeled training data set to learn the task. Instead, it relies on its capability to observe the environment, measure the after-effects of its actions taken on the environment through the rewards, and learn by itself. It does not follow unsupervised learning either, since we, as a programmer already know upfront when our model will learn the task by defining the expected reward. We use the reinforcement learning technique when we have to deal with a reinforcement learning problem that is characterized by a trial and error search along with a delayed reward setting.

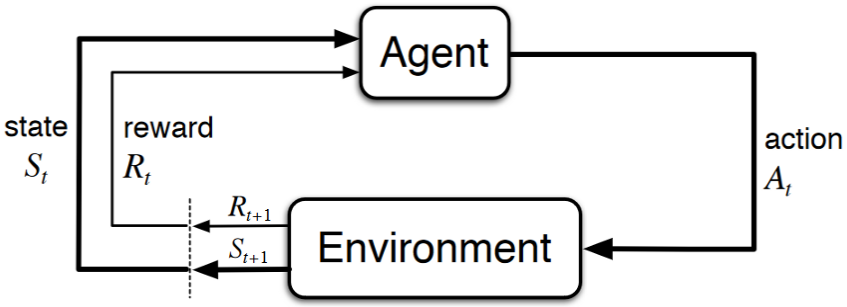


Figure 3.2: The agent-environment action taken from [25, p. 48]]

Oftentimes, reinforcement learning problems denote a complex non-linear system which cannot be modeled easily. Since the reinforcement learning agent learns the task to be achieved in the environment without knowing its model, it is called a 'model-free' approach. Similarly, an agent that knows the mathematical model of the task to be learned beforehand, the approach that it uses to learn is called a 'model-based' approach. In this thesis, we aim at controlling a real-time, dynamical system using model-free reinforcement learning algorithms and therefore, we discuss the reinforcement learning problem further in detail.

The RL problem setup mainly involves three components as seen in figure 3.2. It is due to these components that an AI (or RL) agent is capable of learning a particular task.

1. **The Environment** is the world in which the Reinforcement learning takes place. To exemplify, OpenAI Gym is a standardized toolkit used to develop and compare different reinforcement algorithms. It provides environments for different reinforcement learning tasks such as Cartpole, Mountain Car, Acrobot, etc using which we can train, test, and implement different reinforcement learning algorithms.

2. **The Agent** is the entity that is responsible to execute an action on the environment. The agent is basically the algorithm that can use a neural network and learn the correct way of interacting with the environment.

3. **A Reward function** is the last and most vital part of RL. It is simply a function that maps a certain value to the state of the environment. It returns the value whenever there occurs a change in the state of the environment. Using the rewards, the agent understands whether its action that it provided to the environment in the past was good or bad.

With the description of the important entities in an RL problem, we also enlist the different terminologies that are also observed in the RL problem. They are as follows:

- **Action**: An action space is a set of all valid actions in a given environment. The agent is allowed to take any of the action from the action space at any given time. The action space can be finite (i.e. Discrete) or real-valued (i.e Continuous)

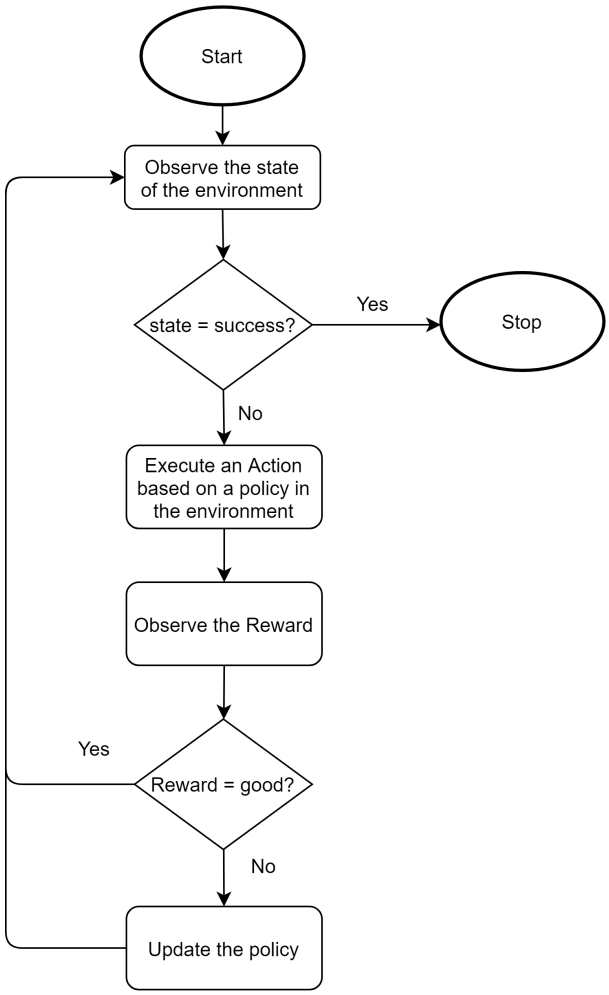


Figure 3.3: A generic flowchart used by RL algorithms to learn a task in a given environment

- **State**: It is a complete description of the world in which the agent finds itself.
- **Observation**: It is a description of the environment and it typically involves a reward associated with the last transition.
- **Transition**: A process through which the environment changes its state to another state when an action is taken in the environment.
- **Return**: The goal of the agent is to maximize the cumulative rewards, called return.
- **Policy**: It is a rule-book that the agent uses to decide how to react to the observed states of the environment. It basically maps the perceived states by the agent to the actions that it is allowed to take in the environment.
- **Step**: A step represents the unit of 'time (t)' in the given environment. So if the state of an environment at a time 't' is S_t, the environment moves to the state S_{t+1} as a consequence of action A_t producing a reward R_{t+1}.
- **Episode**: A sequence of states, actions, and rewards that ends in a terminal or end-state, beginning from a start-state is called an episode. Before the start of an episode, the environ-

ment is reset to an initial state and the agent is allowed to interact with the environment. An episode ends when the environment enters a terminal state. Usually, most of the simulated environments have a predefined maximum episode length to avoid infinite execution time. Thus, an episode can also end if the maximum episode length is reached.

Using the above set of terminologies and concepts, we can write a generic flowchart that every reinforcement learning algorithm follows as given in figure 3.3.

3.2 Deep Q-Learning

Neural Networks (NNs) are a means of doing Machine Learning. They are highly powerful and flexible due to their capabilities of learning a task by analyzing training samples and recognizing patterns in raw sensory data through a kind of machine perception. They are known for their generalization capability from the fact that we do not need to devise an algorithm to perform a particular task. Since NNs are inspired by and loosely modeled on the functioning of a human brain, they are called as 'Artificial Neural Networks (ANNs)'. An Artificial Neural Network consists of an organized set of neurons in layers with an input layer, an output layer, and one or more layers between the input and the output called the 'hidden layers'. Each layer is connected to its adjacent using 'weights'. When an ANN comprises more than one hidden layer, it is termed as a 'Deep Neural Network (DNN)'. Unlike other conventional machine learning algorithms, a DNN performs automatic feature extraction from the data it is provided with, without the human intervention. It assumes that there exists some correlation between the input and the output and produces a function for mapping. It is known as a 'universal approximator', as it learns to approximate an unknown function f(x) = y between any input x and output y. A combination of a Deep Neural Network with a Reinforcement Learning architecture is called 'Deep Reinforcement Learning (DRL)'.

We discussed how an RL agent interacts with the environment in a reinforcement learning setting through executing suitable actions by observing the environment states and improving its policy based on the rewards. The RL agent aims to maximize the return across an episode. In the process of attempting to achieve higher rewards, the agent ends up learning the task. An important aspect of this process is that the state that the agent observes is a consequence of its previous state which in turn is a result of its previous state. Thus, each state within an environment follows a Markov property, i.e. each state depends solely on the previous state and the transition from that state to the current state.

Q-Learning is a model-free reinforcement learning algorithm that uses a Q-table to map all the possible states in an environment to the actions. The 'Q' in the Q-learning represents the q-value i.e. the Quality of taking a particular action when the agent observes a particular state within the environment. Thus, the q-value denotes how useful it is to take a particular action to achieve maximum future rewards. We can estimate the expected future reward from being in a particular state using the Bellman's equation given as,

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

Equation 3.1 states that, the maximum future reward $Q(s, a)$ for an action a taken in a state s is the immediate reward r plus the discounted (γ) maximum future reward for the next state s'. Using this function, the agent can simply follow the policy π and pick the action with the highest q-value,

$$\pi(s) = argmax_a Q(s, a) \tag{3.2}$$

A Q-table is the simplistic model of Q-learning in which we randomly set the q-values for each state-action pair. Based on the q-values, the agent keeps interacting with the environment and iteratively updates the Q-table using the equation 3.1. The values in the Q-table eventually

converge indicating that the agent has successfully learned the problem. An example of a Q-table evolution for an environment with 8 possible states and 4 possible actions is shown in figure 3.4.

At the beginning

States \ Actions	Up	Down	Left	Right
1	0	0	0	0
2	0	0	0	0
3	0	0	0	0
4	0	0	0	0
5	0	0	0	0
6	0	0	0	0
7	0	0	0	0
8	0	0	0	0

After a few episodes

States \ Actions	Up	Down	Left	Right
1	0	0	0	0
2	0	0.12	0	0
3	0	0	0	0
4	0	0	0.55	0
5	0	0	0	0
6	0.22	0	0	0
7	0	0	1.43	0
8	0	0	0	0

Eventually

States \ Actions	Up	Down	Left	Right
1	0	0	0	0.05
2	0	0.18	0.02	0
3	0	0.2	0	0
4	0	0	0.51	0
5	0	0	0	0.01
6	0.31	0	0	0
7	0	0	0.98	0
8	0	0	0	0

Figure 3.4: Illustration of Q-table evolution during the training process.

Thus, we can see that the Q-table serves as a rule-book or a policy or a cheat sheet that an agent uses to solve the task. However, a problem arises when the cheat sheet is too long when an environment has thousands to millions of states. This causes the size of the Q-table to explode unrealistically. At this point, a neural network can be used to replace a Q-table.

A Deep Q-Network is nothing but Q-learning using a Deep Neural Network. While the agent uses a Q-table to pinpoint the best action by looking up their q-values in the Q-table, a Deep Q-network produces the q-values corresponding to all the possible actions when a state is fed at the input of the network. Figure 3.5 depicts the working mechanisms of Q-learning using a Q-table and a Deep Q-Network and highlights how they differ in producing the q-values.

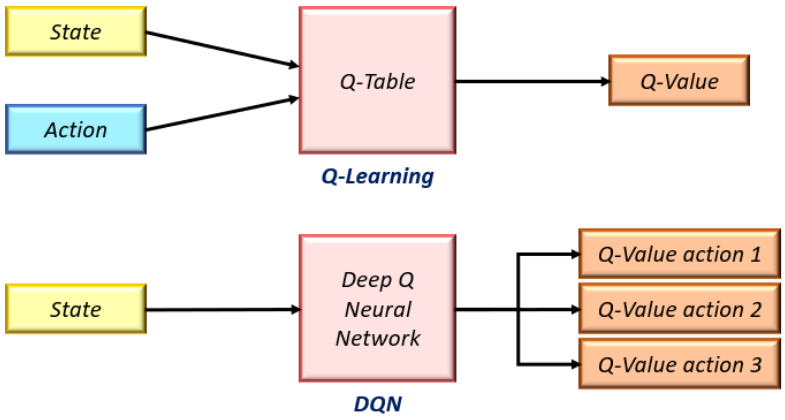


Figure 3.5: Q-Table vs Deep Q-Network.

In 2013, the Deep Q-Network published in [16] proved how a DQN agent could demonstrate the performance of superhuman level on a benchmark of Atari 2600 games. In that, the agent

learned the policies using only the high dimensional sensory input i.e. the raw pixels from the game. The DQN algorithm is as given in listing 1.

Algorithm 1 Deep Q-learning with experience replay [16]

Initialize replay memory D to capacity N
Initialize action-value function Q with random weights θ
Initialize target action-value function $\hat{Q}$ with weights $\theta^{-} = \theta$
for episode = 1, M **do**
    Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi_1(s_1)$
    for t = 1,T **do**
        With probability ε select a random action a_t
        otherwise select $a_t = \text{argmax}_a Q(\phi(s_t), a; \theta)$
        Execute action a_t in emulator and observe reward r_t and image x_{t+1}
        Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
        Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in D
        Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from D
        Set $y_j = r_j$ if episode terminates at step j+1
            $y_j = r_j + \gamma$ otherwise
        Perform a gradient descent step on $(y_i - Q(\phi_j, a_j; \theta))^2$ w.r.t θ
        Every C steps reset $\hat{Q} = Q$
    end for
end for

The working of the DQN algorithm is as follows. The algorithm makes use of a buffer i.e. memory to store the interactions of the agent with the environment. In the algorithm, the action-value function Q represents the Q-network and $\hat{Q}$ also represents the same Q-network but is termed as the Target network. We discuss later how both the terminologies differ in their functionality. At first, the Q-network is initialized with random weights and biases. The agent then observes a sequence of stacked images from the game screen and preprocesses it. This represents the current state s_t. Next, it produces the q-values of all possible actions using the stacked images as the input to the Q-network. With a certain probability, it chooses the action a_t corresponding to the maximum q-value and sends the action to the environment where it is executed. The environment moves to the next state s_{t+1} producing the reward r_t. The DQN agent then stores the tuple (state s_t, action a_t, reward r_t, next state s_{t+1}) into its memory. This process repeats until there are enough samples in the memory to start the training process. Once the memory fills with a certain number of transitions, the agent randomly samples a minibatch of transitions to train the Q-network. With each transition, the next state s_{t+1} is fed to the Q-network with the previous set of weights and biases to produce the 'target q-value'. Since we calculate the target q-value using the previous set of weights and biases of the network, the network is termed as the Target network. It is important to note that the Target network produces the q-values using the previous set of network parameters and not the parameters currently used by the Q-network. Next, the state s_t from the same transition sample is fed to the Q-network to produce the 'predicted q-value'. Then, the loss is calculated using the mean squared error of the predicted q-value and the target q-value as given in equation 3.3.

$$\text{Loss } L = \frac{1}{2}[\overbrace{r + \gamma \, max_{a'} \, Q(s', a')}^{\text{target}} - \overbrace{Q(s, a)}^{\text{prediction}}]^2 \tag{3.3}$$

Using the loss, the network updates its gradient using backpropagation. The agent aims to minimize the loss observed in equation 3.3. The training repeats until the minibatch is completed. The process of learning from the experience of the agent is termed as 'Experience Replay'. Next, the agent again interacts with the environment for one step, stores the transition set, samples a minibatch from its memory, and retrains again. After a fixed set of steps C, the agent copies the

weights and biases of the Q-network to the Target network. The result of the training process ends in the convergence of the Q-network.

3.3 Neural Fitted-Q learning

Martin Riedmiller in his paper [21] introduced a neural network based reinforcement learning algorithm that can learn control policies in a highly data-efficient manner. The proposed algorithm was called as the 'Neural Fitted-Q iteration (NFQ)'. Due to the algorithm's minimal training data requirement, the algorithm could be used on real-control plants. The author of NFQ demonstrated its success on a real-time practical pole swing-up and balancing task in his paper.

One of the major drawbacks of the classical Q-learning algorithms is that they are updating the Q-function in an online fashion i.e. at every transition being added to the memory, the q-value function is updated. Thus, for the agent to be able to learn a task, typically, several ten thousands of episodes have to be played [18]. The main cause for this is, if the weights of the Q-network are tweaked for one particular state-action pair, unpredictable changes are occurring at the other state-action pairs in the state space. This leads to an unreliable and slower learning process [21].

NFQ solves the problem of online training through performing the updating of the Q-network in an offline manner. In that, the NFQ agent initially interacts with the environment (real plant) and collects transitions of the form (s, a, s') where s denotes the current state, a denotes action given to the current state, and s' denotes the next state. This set of transitions describe the experiences of the agent and is called the sample set D. The sample set can then be used to train the NFQ agent in an offline setting using the supervised learning technique. NFQ uses Rprop [19] that is well known to be a very fast and insensitive towards the choice of learning parameters supervised learning technique for batch learning. A pseudo-code describing the NFQ algorithms is as given in listing 2.

Algorithm 2 The NFQ Algorithm [21]

```
1: input: a set of transition samples D
2: output: neural q-value function Q_N
3:     k = 0
4:     init_MLP() → Q_0
5:     DO{
6:         generate_pattern_set
7:         P = {(input^l, target^l), l=1,....,D} where:
8:             input^l = s^l, a^l,
9:             target^l = c(s^l, a^l, s'^l) + γ min_b Q_k(s'^l, b)
10:        Rprop_training(P) → Q_{k+1}
11:        k:= k + 1
12:    }WHILE ( k < N )
```

The NFQ algorithm comprises of two main stages: the generation of a training pattern set P, and training the multi-layer perceptron (MLP) or a Deep Neural Network (DNN) using the training set. The pattern set is divided into input and training sections. The input part is formed using the state s^l and the action a^l from the training experience l. For the cartpole problem, the target value is calculated as the sum of the cost of transition $c(s^l, a^l, s'^l)$ and the expected minimal path costs for the next state s'^l, computed based on the current estimate of the Q-function, Q_k. Finally, using Rprop for fast supervised learning, the network quickly and reliably converges as compared to an ordinary gradient descent technique.

In the case of the cartpole balancing problem, the state space is divided into three regions as shown in figure 3.6. The target value is calculated as,

$$target^l = \begin{cases} c(s^l, a^l, s'^l) & , \text{ if } s'^l \in S^+ \\ C^- & , \text{ if } s'^l \in S^- \\ c(s^l, a^l, s'^l) + \gamma min_b Q_k(s'^l, b) & , \text{ else} \end{cases} \tag{3.4}$$

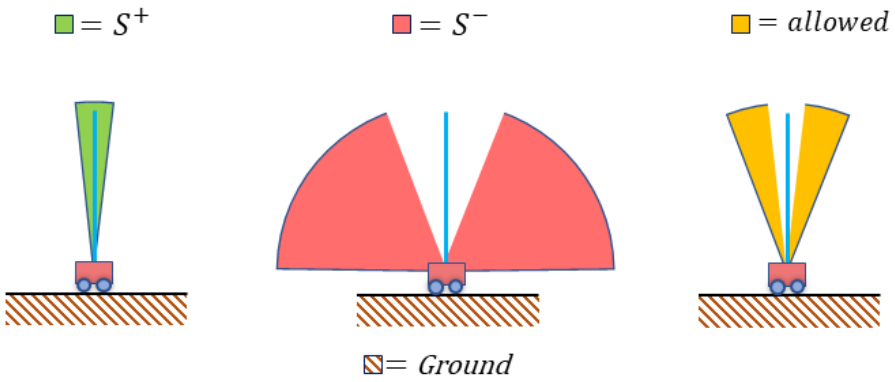


Figure 3.6: Division of the state space in a pole balancing problem using NFQ.

S^+ denotes the set of goal states in which the cartpole system should be controlled to while S^- denotes the set of forbidden states in which the controller being trained must learn to avoid by using a correct control policy [21]. $c(s^l, a^l, s'^l)$ denotes the cost of transition and is set to a small positive constant which indicates that we are aiming for a minimum time controller. In other words, the controller must be able to attain the balancing position at the earliest. Further, we set the target to zero when the system is in the goal region i.e. S^+. And lastly, if the agent enters the S^- region, it has to be punished with the highest degree, hence the target is set to 1 as the maximum output (q-value) generated by the network is 1.

The concept of q-value in the case of NFQ is exactly opposite to that of DQN. In NFQ, the q-value generated at the output of the network represents the cost that it has to incur corresponding to the state-action pair given at the input. Therefore, the aim of the NFQ agent is to minimize this cost by choosing the state-action pair with minimal q-value. This also clarifies why we set the target value highest when a forbidden state is reached and zero when the goal state is reached. The author of NFQ also has provided a guide to applying the NFQ algorithm to practicals systems, '10 Steps and Some Tricks To Set Up Neural Reinforcement Controllers' [20]. We discuss some of the key insights obtained from the guide and enlist the settings used in [21] to implement the NFQ on the real swing-up and pole balancing task.

1. The concept of using the bang-bang controller i.e. oscillating back and forth between two extreme control signals to keep the system in the desired state is often not acceptable in real control plants. Instead, the addition of a neutral action that does not use any energy, to the action space is a better option. However, there exists a tradeoff: a fewer set of actions in the action space leads to coarser control behavior, thereby compromising accuracy in control. On the other hand, multiple higher number of actions can exponentially increase the searching space of available policies. Therefore, the action set should be kept small enough to allow the fast learning of the agent. The paper uses action space A = +F, 0, -F.

2. Choice of S^+: The choice of the goal states S^+ should be such that there exists a policy, that S^+ is reachable from every starting state. With a larger range of S^+, the learning becomes easier whereas a smaller range of S^+ can produce a more accurate controller behavior. In

the NFQ paper, S^+ = cart position is at most 0.1m away from the center AND pole angle is at most ± 0.15 rad i.e. ±8.59°from the vertical.

3. Choice of S^-: The range of S^- used in the paper is ±0.25m from the center.

4. Discounting: To ensure that there exist proper policies through which S^+ is reachable from every state with a non-zero probability, the NFQ paper uses no discounting i.e. $\gamma = 1$.

5. Normalized input: Supervised learning is effective when all the inputs are at a similar level. Thus, the inputs are normalized to zero mean and standard deviation = 1.

6. Epochs: Epochs correspond to the total number of sweeps through the training set and was set to 300 in the NFQ paper.

7. Sampling time: There exists a directly proportional relationship between the sampling time and the number of actions available. With a larger sampling time, more number of actions are required to achieve the same level of controllability and vice versa. Since the action space is small, the sampling time used in the implementation of NFQ by the author is 10 ms.

8. Network architecture: NFQ was implemented using two hidden layers each comprising of 20 neurons with tanh activations and a single neuron at the output layer with sigmoidal activation.

9. Artificial training transitions S^{++}: In order to have a reasonable number of state-action pairs that lead to the goal state, we can add artificial state transitions to the sample set as a part of the hint-to-goal [21] heuristic. Precisely, a hint-to-goal heuristic implies the addition of artificial state transitions, that start as well as end in S^+. These states are denoted by S^{++} and have the terminal cost or target = 0. The NFQ paper applied 3 different artificial transitions namely state (0,0,0,0,0) combined with all three actions. 100 such transitions were added to the training pattern set to establish a balance between a large number of normal transitions and the added special transitions.

10. Exploration scheme: The author of NFQ did not use any explicit exploration scheme. Thus, the Q-function always exploited greedily i.e. the agent chose the action with the minimal q-value and did not perform any random selection of the action to induce exploration of the state space.

Thus, using the settings described above, the target value was calculated as,

$$target^l = \begin{cases} 0 & , \text{ if } s'^l \in S^+ \\ 1 & , \text{ if } s'^l \in S^- \\ 0.001 + min_b Q_k(s'^l, b) & , \text{ else} \end{cases} \tag{3.5}$$

3.4 Genetic Algorithm

In the field of Artificial Intelligence, Evolutionary Algorithms (EAs) are mainly used for the optimization of existing solutions for a problem. However, they can also be used as standalone algorithms to find optimal solutions from scratch to a problem through randomized search heuristics. EAs are inspired by the biological evolution and are based on the evolutionary theory of Charles Darwin, "Survival of the fittest". An EA is an optimization algorithm that mimics the biological mechanisms such as mutation, recombination, and natural selection to find an optimal design within specific constraints [9]. EAs differ from the other optimization algorithms from the fact that they are dynamic in nature and they 'evolve' over time.

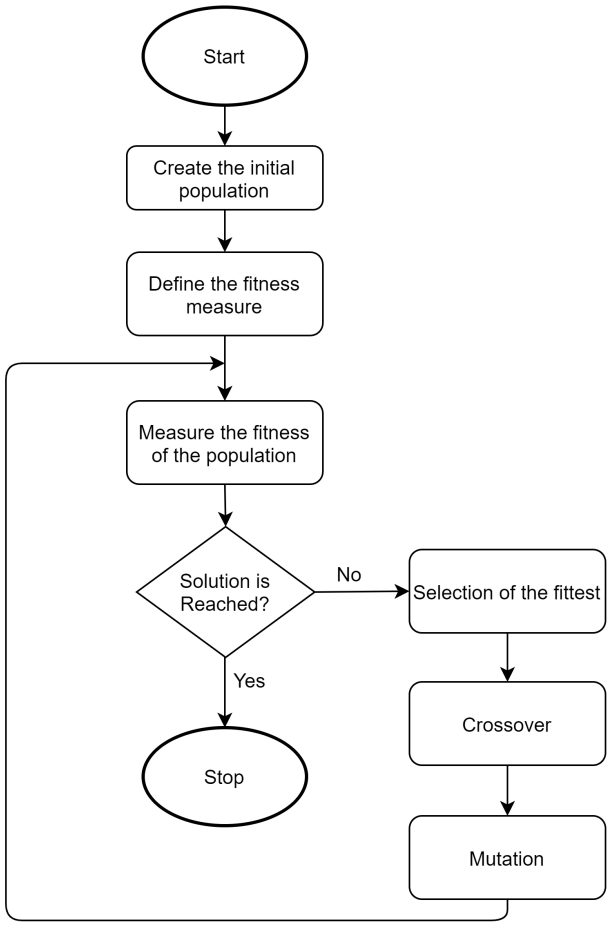


Figure 3.7: Working mechanism of the Genetic Algorithm

Genetic Algorithm [14], first proposed by John Holland is one of the most popularly used EA. GA is different from RL despite they both solve the same class of problem: maximizing or minimizing a reward or a cost function. The main difference lies in the methodologies that each of the algorithms uses. In GA, the algorithm ends in finding the most optimal solution for the given task. On the other hand, RL is a continuous long term learning process of an agent through the exploration of the state space of the environment. Oftentimes, GA is used on top of RL to improve the solution provided by the RL algorithm. The Genetic Algorithm's workflow is given in figure 3.7 using which we understand the mechanism of GA in depth.

GA is a population-based optimization algorithm, meaning that a population of sub-optimal or bad solutions to a problem is used to produce better-optimized solutions. Hence, referring to the flowchart given in 3.7, we start with generating a population in which each solution is termed as an 'individual'. Each individual is represented using a set of features, called 'chromosomes'. And each chromosome contains a set of 'genes'. Since we consider the optimization of the neural network controller in this thesis, consider the example of a neural network as an individual in a population. Figure 3.8 describes the population setup in a GA and also maps it to an example of a neural network as an individual for optimization.

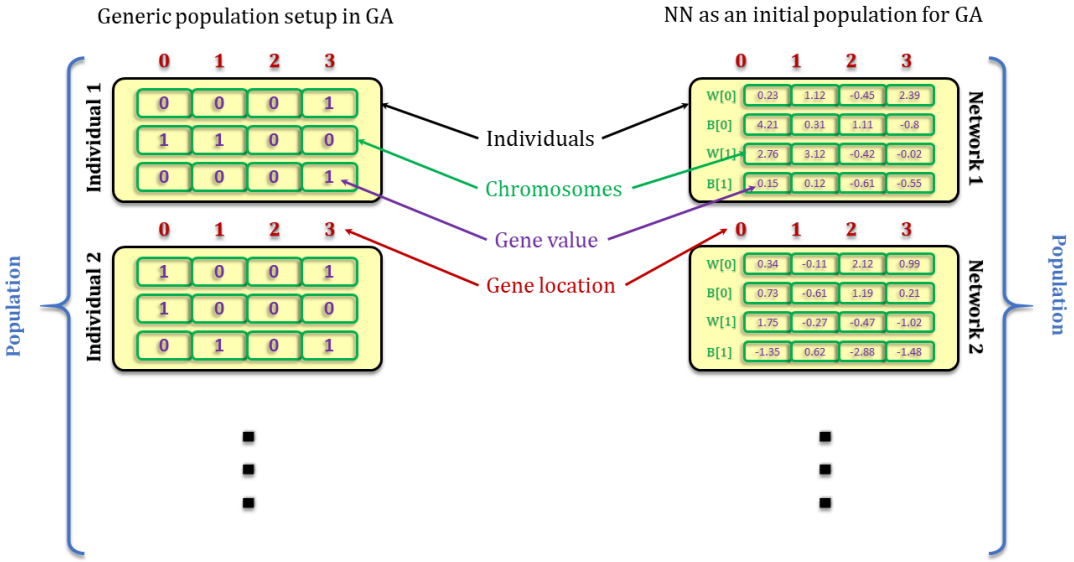


Figure 3.8: Mapping the population setup of GA to that of a Neural Network population.

The population size is defined at the initial stage of the GA. When using neural networks for optimization, each neural network represents an individual with the weights and biases (shown as W[i] and B[i] in the figure 3.8 which denote the weights and biases of the i^{th} layer) as its chromosomes. Every value of the parameter of the neural network from the weights and biases forms a gene located at a specific location. Once the population is set up, the next step is to define a fitness function.

The fitness function is used to calculate the fitness value of an individual in the population. In control applications like a cartpole system, we use the controlling capability as the fitness function. For example, the maximum balance time achieved by a neural network i.e. an agent can be set as the fitness function. Thus, the higher the fitness value achieved by the network, the better is the quality of the network. Defining the function is an essential part of the GA as it helps to evaluate the quality of a solution in a population.

The next stage is calculating the fitness value of each individual. Therefore, in the cartpole problem, each network is tested on the system and the performance of the network in terms of balance time can be noted as its fitness value. If the fitness values of the desired individual or the entire population are not optimal, we proceed for the next step that is inspired by the biological process of 'reproduction'.

To perform reproduction, first, we select the individuals that have the highest fitness scores. These individuals are treated as the 'parents' for generating the offsprings of the next generation of the population. This process helps in slowly killing individuals with bad qualities within one generation of the population and gradually producing better individuals using good quality parents

for the next generation. In the case of neural networks, the top best performing AI controllers on the real-cartpole system are selected. Next, we perform a process called 'Crossover' that mimics the mating process in biology. In this, the genes in the chromosomes of each parent are fused to produce the chromosomes of the offspring. This process is illustrated in figure 3.9. The amount of genes corresponding to each parent into their offspring is random. As seen in the illustration, the offspring has sometimes carried an equal number of genes from each of its parents thus inheriting the qualities of its parents. In the case of a neural network, the offspring shares the weights and biases from each of its parent networks randomly. Crossover mechanism is an important step in GA as it ensures that the offspring will be different from its individual parent.

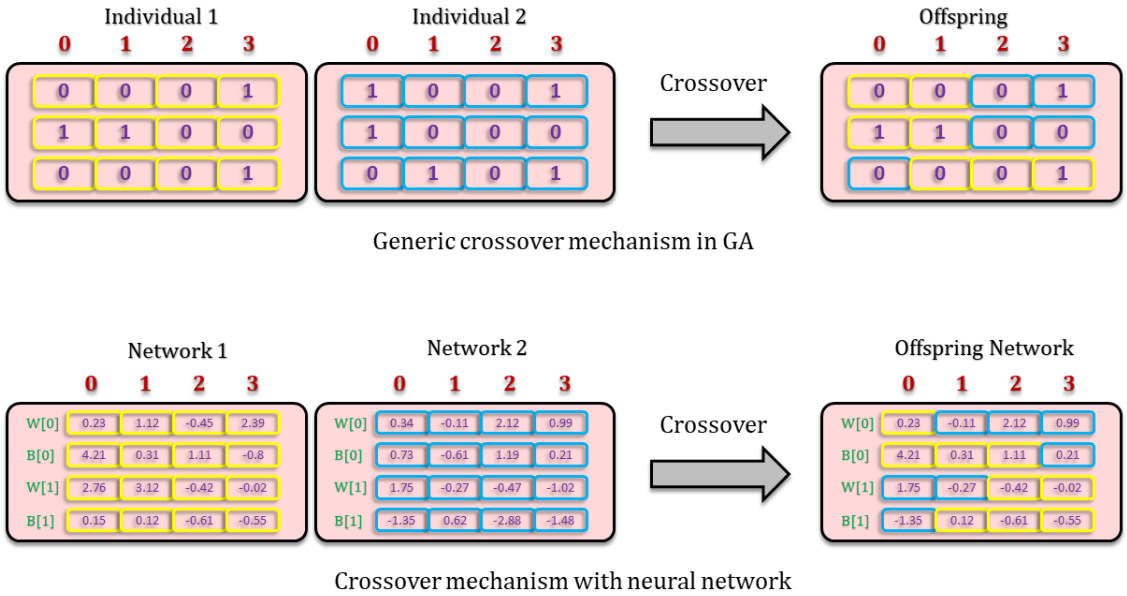


Figure 3.9: Illustration of crossover mechanism in GA.

Finally, the last key component of GA that is responsible to bring diversity in the population is the 'Mutation' operation. Using mutation, we change the genes in the offsprings with a mutation rate. In general, if the genes are represented by bits, then the mutation process can result in random flipping of bits to induce variation in the offspring. From the ML point of view, mutation helps to avoid getting stuck with a solution in the local minima of the solution space. When neural networks are mutated, the values of the weights and/or biases get tweaked according to the mutation rate used. The mutation mechanism is illustrated in figure 3.10.

A commonly used strategy to produce better solutions in every generation is 'Elitist selection'. It implies that every time a new generation is reproduced using a set of parents, one or more elite parent is carried over to the next generation, unaltered. A parent is considered elite if it is found to be the fittest from its generation. Similarly, in the case of neural network optimization using GA, the network with the highest score from the previous generation is carried over to the next generation with regards to elitism.

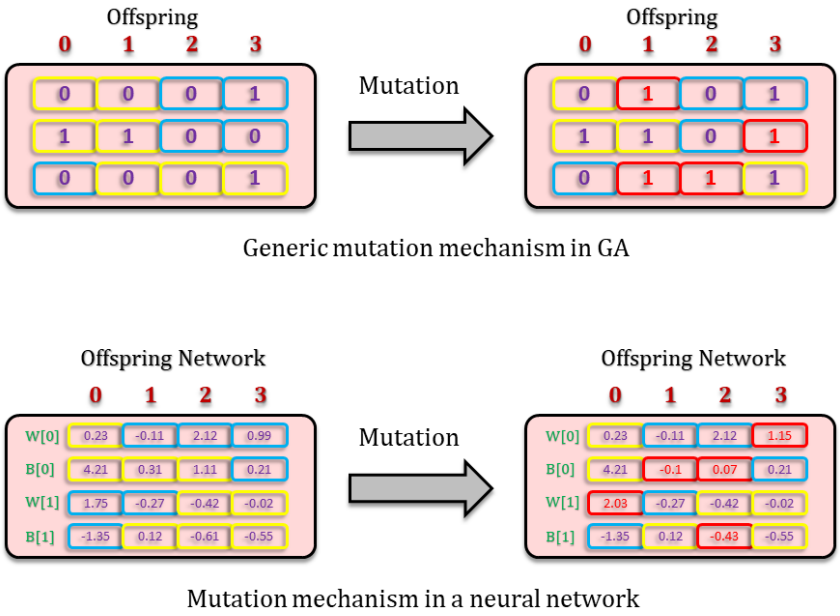


Figure 3.10: Illustration of crossover mechanism in GA.

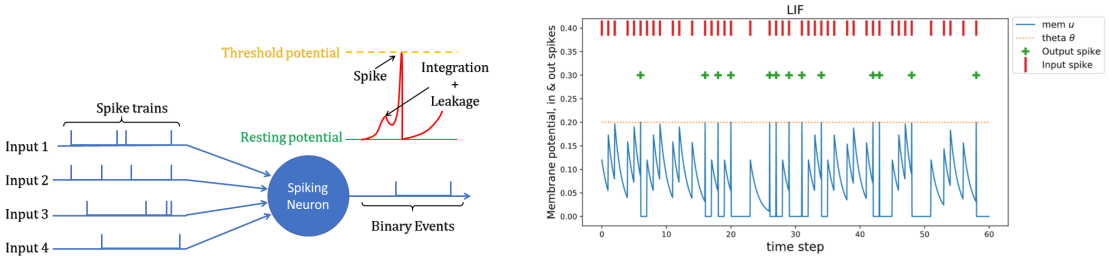
3.5 Spiking Neural Networks

Despite the prosperity of Artificial Neural Networks in the field of ML for reportedly being analogous to a human brain, they are biologically inaccurate and do not exactly mimic the functioning of a biological neuron. The closest representation of the functioning of a biological neuron is the Spiking Neuron that was proposed back in 1952 by Hodgkin and Huxley [13]. To comprehend the working mechanism of a spiking neuron, we describe how a biological neuron works in brief.

Owing to the biochemical constitution of a human brain, a biological neuron within our brain has unequal concentrations of ions on the outside and the inside of its membrane. This difference in the concentrations of ions gives rise to a potential difference. The neurons within a human brain communicate with each other through voltage spikes. Based on the electrical theory and concepts, a voltage spike can be produced with a battery with a switch closed for an extremely short period and opened immediately. A biological neuron produces a spike or in biological terms, fires an 'action potential', when it receives sufficient stimulus. This action potential travels through the axons of the neuron until the terminal where it releases neurotransmitters. These neurotransmitters get attached to the receptors of other neurons which causes a change in the membrane potential of that neuron. Depending on the type of neurotransmitters, the potential difference across the membrane of the neuron may increase (for excitatory neurotransmitters) or decrease (for inhibitory neurotransmitters). Whenever the membrane potential of a particular neuron reaches a certain threshold, it fires the charges contained inside it, thereby dropping the membrane potential drastically. A neuron is connected to another neuron via a 'synapse'. On average, a single neuron is connected to tens or hundreds of other neurons. The voltage spikes are transmitted through the synapses to the other neurons and the strength of this synapses decides how much influence it has over the neuron it is connected to. The strength of a synapse translates to the number of neurotransmitters that a neuron releases. With a neuron being connected to multiple other neurons through their respective synapses, the net effect of voltage spikes received from all the neurons adds up to decide when the current neuron will fire. The neurons that send voltage spikes to a certain neuron are called presynaptic neurons while the neuron on the receiving end is called the postsynaptic neuron. Once a neuron fires its action potential or the voltage spike, it returns to a stable state and the potential inside the membrane resets. The process of receiving

spikes and changing the membrane potential repeats forever and this demonstrates the functionality of neurons within a human brain.

Spiking neurons are inspired by the biological neurons and are represented using a Leaky-Integrate-and-Fire (LIF) model. The LIF model is one of the most basic and widely used neuron model. In that, each neuron has a predefined threshold voltage and a membrane potential. With each spike incoming from input neurons, the membrane potential of the neuron increases until it reaches a threshold point as seen in figure 3.11a. Once the threshold voltage is reached, a spike is produced at the output of the spiking neuron, and the potential of the neuron is reset to the resting potential. The spiking neuron demonstrates a small potential dissipation i.e. leakage of the potential to the environment as shown in figure 3.11b. When a train of input spikes (colored in red) forces the membrane potential θ (line graph in blue color) of the Spiking neuron to increase, the neuron produces a spike (shown by a '+' colored in green) at its output by dropping its potential to a stable resting state.



(a) (b)

Figure 3.11: (a) LIF spiking neuron working mechanism, (b) LIF spiking neuron behaviour taken from [28]

The LIF neuron is mathematically represented using the equation 3.6 [10].

$$\tau_m \frac{du}{dt} = -[u(t) - u_{rest}] + RI_t \tag{3.6}$$

The LIF neuron integrates the input current $I_t = \sum_{t_i} \delta(t_i)$ expressed in terms of a spike train $t(i)$. The term $\tau_m = \text{RC}$ is a time constant and it represents the leaky nature of the LIF neuron. R represents the resistance of the neuron's membrane. Thus, the RHS of the equation 3.6 represents the leaky integration of input signal of the spiking neuron. The LIF neuron fires a spike s_t when its membrane potential given by $u(t)$ crosses the fixed threshold denoted by θ. Thus, the spike can be modelled as a non-linear function given by,

$$s_t = \hat{f}_s(u(t), \theta) \tag{3.7}$$

Finally, the membrane potential $u(t)$ resets to a resting potential u_{rest} once the spike is fired and given by,

$$u(t) = u(t)(1 - s_t) + u_{rest} s_t \tag{3.8}$$

In summary, the difference between the working of an SNN and an ANN is that the SNN does not fire each neuron for every feedforward path of the network but rather fires only when the membrane potential of its neuron reaches a certain value. This effectively produces fewer spikes at the output of an SNN for a given number of feedforward iterations as compared to that of an ANN.

In day-to-day life, human brain encounters activities that are mostly sequential. SNNs have adopted this behaviour through the memory that each LIF neuron maintains. As a result, SNNs

can perform well even when we feed them individual frames of sequential data one at a time. On the contrary, ANNs require a set of consecutive frames of temporal data in order to control sequential control systems as they do not possess memorizing capabilities inherently. This makes SNNs suitable for applications that are sequential and need to be controlled. On the similar lines, when static applications such as image classification are concerned, ANNs can perform better as they do not depend the previous input that was classified.

Finally, SNNs work on the principle of transmitting and receiving binary signals (0 or 1) as opposed to ANNs that communicate continuous values. This makes SNNs more robust to noise as the binary value still gets communicated as a spike or a no spike. On the other hand, in ANNs, a continuous value added with noise that is transmitted to the next neuron can be considered meaningful as the next neuron directly uses the received signal for further calculation.

Chapter 4

Evaluation Framework

In this chapter, we provide reasons for the choice of the hardware used to build a real cartpole system. We begin with a feasibility study and describe the market survey we conducted to buy existing cartpole system. Next, we show the blueprint of the expected cartpole system based on which we built the cartpole system using LEGO Education kit. The specifications of the designed cartpole system and the component characteristics are tabulated. Next, we describe the software setup used to communicate with hardware and implement AI controllers on it. We chose to use an on-board mode of operation based on the delays observed in the hardware and the results are described. Owing to the choice of on-board computation, we describe the PyBrain API that we used to implement the AI controller on the hardware.

4.1 Feasibility Study

Since we plan to implement the AI controllers on a real system, we studied the possible options to build or buy such a system. Upon researching for existing solutions, we found that there are companies that provide a variety of plants to execute control over a single inverted pendulum (or a single-pole cartpole), rotary inverted pendulum (single or double), etc. Quanser[1] is one such company that provides several solutions such as an Inverted Pendulum with a Linear Servo Base Unit, a Rotary Inverted Pendulum module attached to a Rotary Servo Base Unit, and a QUBE – Servo 2 which is a low-cost teaching platform with fully integrated, modular servomotor. Bytronic Educational Technology[2] provides a Pendulum Control System (PS2) that can function as a stand-alone system that can be used as an inverted pendulum as well as an overhead crane. Asti Automation[3] is another company that provides a similar solution with an AC servo motor and inbuilt sensors. All the mentioned companies provide their respective products along with the licensed Matlab Simulink software experiment platforms and all the necessary subsystems such as voltage amplifiers, data acquisition, signal conditioning, etc to support the cartpole system. The above systems cost from €5000 to €12000 for an entire full-fledged system.

Owing to the limited budget, we could not buy any of the comprehensive set of systems as described above. Therefore, we sought for options to build a cart-pole system on our own. We found a possible solution of LEGO EV3 Core Set provided by LEGO Education which comprises of an EV3 controller (also called as Py-Brick or just brick), a set of motors, sensors and several modular parts that can be arranged together to make a movable four-wheeled cart powered by a rechargeable battery. We planned to install a pole on top of this cart to build our desired cartpole system. One advantage of building such a system using LEGO is that we could make modifications to the system dynamically, for e.g. changing the pole length, adding weight to the pole/cart more

[1] See https://www.quanser.com/ for details.
[2] See http://www.bytronic.net/ for details.
[3] See https://astiautomation.com/ for details.

easily as compared to buying parts of the cartpole with a fixed length and weight. The basic EV3 core set along with the required supplementary parts was purchased at a price of €418 leaving some space to buy additional parts if required.

Alternatively, we could also choose to buy separate parts of the cartpole system such as a linear guide rail to provide linear sliding motion for the cart, a gyroscope, and a pole, a linear motor controller, voltage amplifier, etc. Apart from the fact that the cost of building this system would slightly be over the budget, the end product would result in a system fixed to some rigid support (such as a table) which inhibits the mobility of the system as a whole. Since we are aiming to build a demo product to perform our experiments, it is advantageous to make it flexible and mobile. Further, building such a system is more time consuming and complex as compared to that of a system built using LEGO. A cartpole system built through LEGO is not fixed to a surface and is easily portable. We can further extend the communication of the hardware (EV3 brick) through either USB communication or via Bluetooth. This makes our product more mobile and hassle-free. Therefore, we preferred to build a four-wheeled, EV3-controlled cart mounted with a set of motors and sensors and a pole on top to realize the real cartpole system for our project.

4.2 Hardware setup

To design a Cartpole using LEGO EV3 Educational kit, we required the following components:

1. *An EV3 brick* (controller): This is the central controller on the hardware responsible for communicating with the sensors and motors and sending commands to perform actions as required. Its specifications are as shown in table 4.1

Hardware specifications for the EV3 Programmable brick	
Main processor	32-bit ARM9 processor 300 MHz OS: LINUX
Memory	64 MB DDR RAM 16 MB FLASH 256 KB EEPROM
Micro SD-Card interface	SDHC standard, 2 – 32 GB
Bluetooth	Bluetooth V2.1 EDR Primary usage, Serial Port Profile (SPP)
USB 2.0 Communication, Client interface	High speed port (480 MBit/s)
USB 2.0 Communication, Host interface	Full speed port (12 MBit/s)
4 input ports	UART communication Up to 460 Kbit/s (Port 1 and 2) Up to 230 Kbit/s (Port 3 and 4)
4 output ports	6 wire interface supporting input from motor encoders
Power source	Rechargeable Lithium Ion battery, 2000 mAH
Connector	6-wire industry-standard connector, RJ-12 Right side adjustment

Table 4.1: LEGO MINDSTORMS EV3 hardware specifications[8]

2. *Two EV3 large motors*: It includes tacho functionality with a resolution of one count/degree enabling accurate speed control of the motor. The specifications are as given in the table

4.2

Operating condition	Characteristics
9 volt, no load	175 RPM 60 mA
Stalled	40 Ncm 1.8 A

Table 4.2: LEGO MINDSTORMS EV3 Large motor specifications[8]

3. *An EV3 medium motor*: It also includes tacho functionality with a resolution of one count/degree enabling accurate speed control of the motor. The specifications of the medium motor are as indicated in the table 4.3

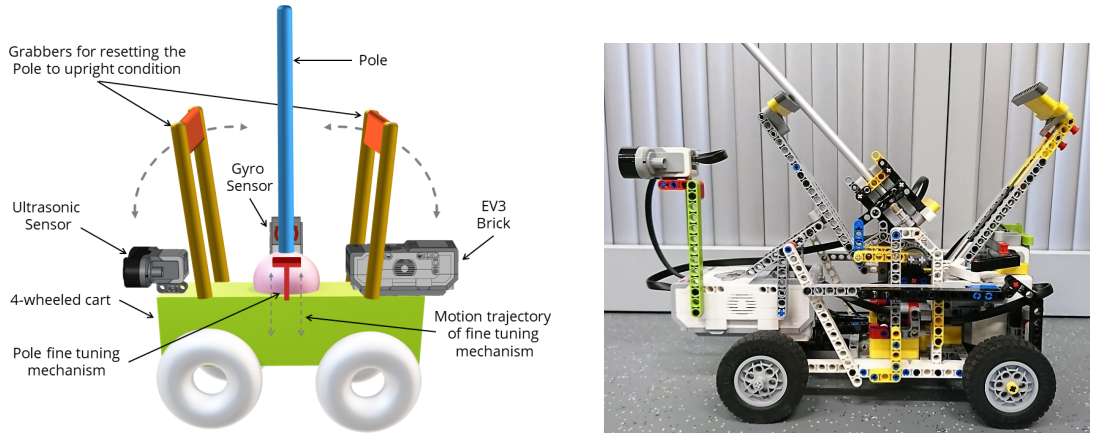
Operating condition	Characteristics
9 volt, no load	260 RPM 80 mA
Stalled	15 Ncm 800 mA

Table 4.3: LEGO MINDSTORMS EV3 Medium motor specifications[8]

4. *An EV3 Gyro sensor*: It operates at a sampling rate of 1ms if the sensor reading changes fast enough. If not, the sensor is programmed to return a new sensor value automatically whenever the value changes, or when the EV3 brick requests the latest sensor value. The Gyro Sensor supports up to 440°/sec with a accuracy at ± 3° within 90°. The Gyro sensor can operate in the following three modes:
   a) 'GYRO-ANG' Mode: Returns the number of degrees the sensor has been rotated since it was switched to this mode.
   b) 'GYRO-RATE' Mode: Returns the rate at which the sensor is rotating, in degrees/sec.
   c) 'GYRO-G&A' Mode: Returns the angular velocity as well as the angle measurement.
5. *An EV3 Ultrasonic sensor*: The ultrasonic sensor measures the distance to the object by sending out an ultrasonic sound signal, with a sound burst of 12 at 40 kHz. It can measure up to 250 cm (100 Inches) with an accuracy of 1 cm (0.393 Inches) in either centimeters or inches. The sensor values are returned with a 0.1 resolution.
6. *LEGO Bricks*: Apart from the LEGO building bricks provided in the set MINDSTORMS EV3 Core Set (45544), we required additional parts which were taken from the LEGO MINDSTORMS EV3 Expansion Set (45560).
7. *Four wheels with rim*: LEGO MINDSTORMS EV3 Core Set (45544) provides only one pair of large wheels with rims. Since we planned to build a four-wheeled cart, we had to use four large wheels and rims from the LEGO MINDSTORMS EV3 Expansion Set (45560).

The idea was to build a four-wheeled cart controlled by the EV3 brick and two EV3 large motors. However, on experimenting, we observed that only one large EV3 motor was sufficient to move the cart along with all the parts and pole mounted on top. Hence, a large motor was attached to both the rear wheels making it a rear-wheel-drive (RWD) cart. Since we have to perform training of the neural network in an episodic fashion, the pole needs to be brought to an initial position if it exceeds the limit within which it should be balanced. By resetting the pole,

we mean that the pole is made upright such that its angle from the exact vertical is zero. We automated this process by designing a pair of grabbers that bring the pole back to the upright position whenever an episode ends. The structure of the planned cartpole mechanism is as shown in figure 4.1a. The grabbers are designed such that they follow the trajectory shown by the dotted curves. We have used the second large motor to operate the grabbers. Though the grabbers were intended to make the pole upright, we had to design another mechanism to fine-tune the pole to make it almost exactly vertical and perform the releasing action before the start of the next episode. To implement this mechanism, we used a medium EV3 motor which holds the pole upright while the grabbers move outwards. An EV3 Gyro sensor is attached to the axle around which the pole rotates. Thus, the gyroscope also rotates in alignment with the pole movement. Lastly, we use an EV3 Ultrasonic sensor on the front of the cart to measure the distance traveled by the cart.



(a) Cartpole mechanism blueprint

(b) Actual Cartpole System

Figure 4.1: Hardware components of the Cartpole System

The actual cartpole system was built as depicted in figure 4.1b. We have added a few more pictures of the final assembly in appendix B. The specifications of the designed cartpole system are described in the table 4.4.

Sr. No	Parameters	Values
1	Cart Dimensions (w x l x h)	17 x 36 x 25 cm
2	Cart weight	1.93 Kg
3	Pole weight	0.052 Kg
4	Overall weight of the system	1.982 Kg
5	Pole height from ground	14 cm
6	Pole Length	1 m
7	Wheel diameter	6.24 cm
8	Wheel thickness	2 cm
9	Maximum speed	3.66 m/s
10	Maximum acceleration	7.208 m/s2
11	Maximum force	14.29 N

Table 4.4: Specifications of the Cartpole system

4.3 Software setup

EV3, the brain of the LEGO Mindstorms, has a Linux computer at its core. We can, therefore, program it using several programming systems such as Python, C++, C, Java, or Prolog apart from the EV3 Programming app which is the official programming app from LEGO Education. We choose to program EV3 using python as we would be implementing a neural network on the controller which is designed using the python programming language. EV3 Python is an extended version of the python programming language that allows using python to control the LEGO EV3 robot. EV3 Python needs to run on top of the EV3dev which is a Linux based operating system. To boot up the EV3 brick with EV3dev, we simply download the EV3dev image file into a microSDHC card (2GB or larger), insert it into the EV3 brick, and start the brick. Once the EV3 has booted up, we have one of the following options to connect the EV3 to our PC using a technology called 'tethering[4]':

(a) Using Bluetooth

(b) Using USB cable

We choose to connect EV3 to the PC using Bluetooth at the initial stage to provide a basic wireless setup for experimentation. Once the Bluetooth connection is set up, we need to connect to the EV3dev using SSH so that we can run commands on the EV3 to deploy codes, make changes in setting, etc. remotely, over the network. Among many applications available such as PuTTy, OpenSSH, etc. we used MobaXterm toolbox to establish the SSH connection as it features a window displaying folders on the remote device. This completes the connection setup and scripts can be written on the EV3 remotely. We used Visual Studio Code as the IDE to write the python programs on the PC and then either download or remotely run the scripts on EV3.

Running simple python scripts such as rotating a motor or performing basic operations on a cart (such as move front or back) using EV3 is an easy task as it involves less computation. Here we are considering that the python script is residing and running on the EV3's ARM9 processor and we term this as 'on-board' computation. Complex tasks such as performing training of a neural network on the EV3 can be too heavy for its processor. As a result, we had to find a solution to run the programs remotely on the EV3 without the actual python program residing on the EV3 brick as opposed to 'on-board' computation.

4.3.1 RPyC

Remote Python Call (RPyC) is a python library that is used for making remote procedure calls, clustering, and distributed-computing. It uses object-proxying, a technique that employs python's dynamic nature, to overcome the physical boundaries between processes and computers, so that remote objects can be manipulated as if they were local [3]. RPyC mainly features Transparency, meaning that it provides access to remote objects as if they were local and, a Symmetric protocol by which both, the client and the server can serve requests. To start using RPyC, we must install RPyC both on the EV3 and on the PC.

Issues while using RPyC

It is important to use the same versions of RPyC on both EV3 and the PC. Initially, we followed the instructions given in [4] which resulted in installing RPyC version 4.1 on PC while version 3.3 on the EV3. This caused the version discrepancy issue. We then tried to solve this issue by installing RPyC version 3.3 on PC as well. Despite both the devices having the same versions this time, we still ran into another issue. On investigating, we found out that there is a file named "async.py" and it is referenced by many other files in the RPyC directory. However, "async" was introduced as a keyword between RPyC version 3.3 and RPyC version 4.0. To troubleshoot this

[4]This process does not provide Internet access to the EV3 brick after connection.

problem, we changed all references from “async” to “async_”. This got the RPyC working normally on both the devices.

An important disadvantage of using RPyC is the latency introduced by a network connection. Though RPyC aids in using PC’s computational power (RAM, GPUs, etc) instead of EV3’s resources for training a neural network, every command to be executed by the EV3 comes with a round-trip delay over the network connection. It is therefore crucial to analyze this round-trip latency and check if it is suitable for our cartpole system. In case of excess delay, we might have to opt for a wired connection i.e. using USB to connect EV3 to the PC.

4.3.2 Delays

A simple ‘if-else’ control logic of moving the cart in the forward direction if the pole has inclined in the forward direction (i.e. Pole angle $> 0°$) and vice versa was used to calculate the delays. We repeated this ‘if-else’ control logic 500 times and calculated the average amount of time required to produce action by three possible means: On-board, USB, and Bluetooth. The results of the experiment are given in table 4.5 which represent the average time and variance for a single iteration of ‘if-else’ loop.

Mode of operation	**Avg. computation time (in ms)**	**Variance (in ms)**
On-Board	16.9451	0.0103
USB	152.600	0.561
Bluetooth	235.648	1.305

Table 4.5: Average computation latency for different modes using EV3

The results underpin the intuition that performing on-board calculations took the least amount of time whereas the average computation latency increased as we shifted from wired to wireless communication. By wireless communication, we mean that the data is read through the sensors by the EV3 brick, transferred over the network to the PC where the next motor action is calculated, and the action is sent back to the EV3 brick. Every command that is sent to/from the EV3 brick adds a round-trip delay to the overall latency which makes Bluetooth communication temporally most expensive.

Further, we can see that on-board computation was around 10 times faster than USB and 14 times faster than Bluetooth modes of operation. We had performed experiments on simulation of ‘Cartpole-v1’ environment using OpenAI Gym for maximum allowable sampling time to be able to successfully balance the pole upright. The latencies of the Bluetooth and USB modes of operation are too large to be suitable for use in a real-time setting of the cartpole problem. Therefore, we chose on-board mode of operation for the cartpole problem. It also necessitates research for different options to implement a neural network on the EV3 brick (i.e. on-board) as it nullifies the chances to use wired or wireless modes of operation to control the cartpole system.

4.3.3 PyBrain

Keras is one of the leading open-source neural network libraries. It uses TensorFlow at its backend and provides several advantages such as consistent and simple APIs, lowering the number of user actions needed for typical usage cases thereby delivering reduced cognitive load [26]. Therefore, we preferred to use Keras as our Deep Learning API to design and evaluate our neural network. Since EV3 brick has a very small memory (RAM = 64 MB DDR) it was not possible to install Keras or any other popular Deep Learning APIs such as Pytorch or TensorFlow on-board since they required better system requirements. This paved way for a modular Machine Learning Library

for Python called PyBrain [24].

PyBrain aims at providing flexible, easy-to-use yet powerful algorithms for neural networks, reinforcement learning, supervised learning, unsupervised learning, and evolution. Mainly, it could be installed on the EV3 brick which enabled us to run a neural network on the hardware. However, simply using PyBrain to run a network trained on Keras was not straightforward. We followed a systematic approach to substitute Keras by PyBrain as described in the following section.

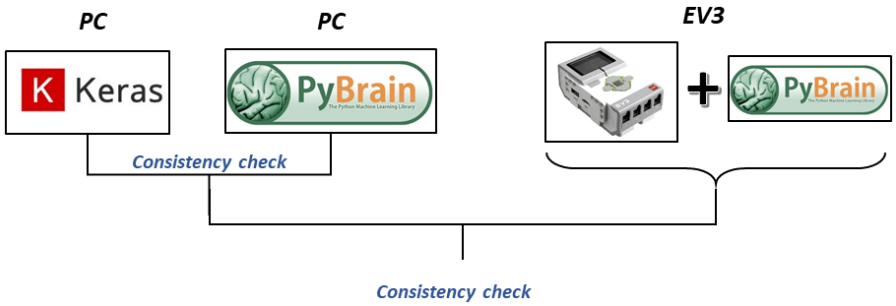


Figure 4.2: Consistency checks required to successfully substitute Keras with PyBrain on-board

The aim was to obtain consistent results i.e. output from the neural network when both Keras and PyBrain were used. Therefore, we had to perform two consistency checks as seen from figure 4.2. First, we considered checking the consistency of using both the APIs on simulation rather than directly testing on EV3 brick. Thus, we installed PyBrain on PC. Next, we built a network (or model) using Keras and the model weights and biases were saved in HDF5 binary data format as it is the default format used by Keras. The saved .HDF5 model had to be converted to .xml format as PyBrain can only load/save a network in this format. To perform the consistency test, a fixed set of input values were provided to both networks i.e in Keras as well as in PyBrain libraries and the output values were compared. Initially, we observed a discrepancy in the results as they were not matching despite the weights, biases, and input to the network were exactly the same.

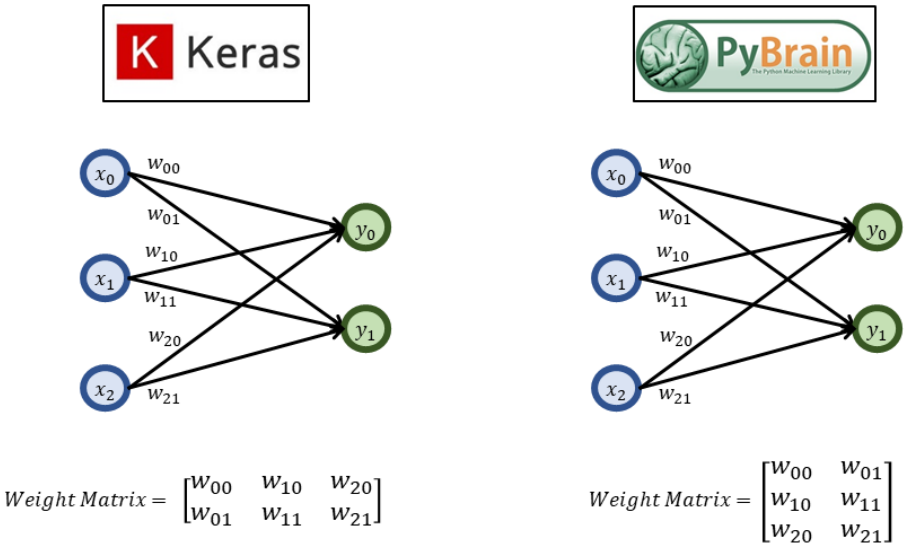


Figure 4.3: Methodology used by Keras (left) for storing weights (Row-major Order) and PyBrain (right) for storing weights (Column-major Order)

After digging into the discrepancy, we observed that the issue was with the manner in which the weights and biases are stored by Keras and PyBrain. From figure 4.3, we can see that Keras

stores weights (and biases) in a row-major order whereas PyBrain stores weights in a column-major format. Due to this, the network was producing inconsistent results. We then corrected the manner in which the model was written from .HDF5 format to .xml format and were able to observe consistent results on PC. The corresponding code to write the weights from .HDF5 file to .xml is uploaded to the GitHub[5]

Lastly, we cross-checked if the exact same results were obtainable from running the network on EV3. We could see the consistency in the results in this experiment as well and therefore, chose to use PyBrain for running a neural network on the hardware.

[5] https://github.com/yaskam8/Thesis_work

Chapter 5

Implementation: DQN

This chapter discusses the experiments that were conducted to obtain a DQN agent that could balance the pole on the hardware. It explains various steps that started with obtaining a DQN agent through simulations and further on the hardware. We discuss why the implementation of the online DQN is not straightforward in the practical world and propose an offline training approach to train the DQN agent. The chapter also covers major modifications in the software setup that were used to make the real cartpole system feasible to be controlled by a DQN agent. Another main finding of the chapter is the existence of a reality gap that causes the failure of simulated DQN agents to control a real hardware setup of the cartpole system. In the end, we describe how we obtained a DQN agent through offline training that could balance the pole for an average of 2.2 seconds and a best performance of 3.7 seconds.

5.1 Motivation

Deep Q-Network (DQN) had gained popularity due to its super-human performance on a number of Atari games. It is one of the fundamental algorithms in the world of reinforcement learning and has been the anchor for many recently developed advanced RL algorithms. We chose the DQN algorithm to implement on our real-life cartpole system due to its simple and powerful nature.

5.2 Experiments

DQN was implemented on a high dimensional sensory input i.e. pixels of images in [16] to demonstrate its superhuman performance in Atari 2600 games. Thus, it involved convolutional layers followed by pooling in the architecture of the network. In our case, we provided a low dimensional sensory input obtained from Gyro and Ultrasonic sensors installed on the hardware. As a result, the state that described our environment i.e. the real-life cartpole system is: the cart position x from the center of the track measured by the ultrasonic sensor, the cart velocity $\dot{x}$ calculated using the ultrasonic sensor, the pole angle θ measured using the gyro sensor, and finally the angular velocity $\dot{\theta}$.

DQN algorithm is an online training approach that interleaves interacting with the environment and learning policy. To train a DQN agent (i.e neural network), at every step, the reward from the cartpole system needs to be sent back to the agent as shown in figure 5.1. The agent then undergoes training where its weights and biases are modified. Next, the updated agent is now used to calculate the next action and this cycle is repeated until the agent gets trained to completely be able to balance the pole.

Balancing the cartpole is an inherently real-time task; neither an arbitrary amount of time is available in choosing an action nor backtracking is feasible [22]. This is non-trivial as we have to

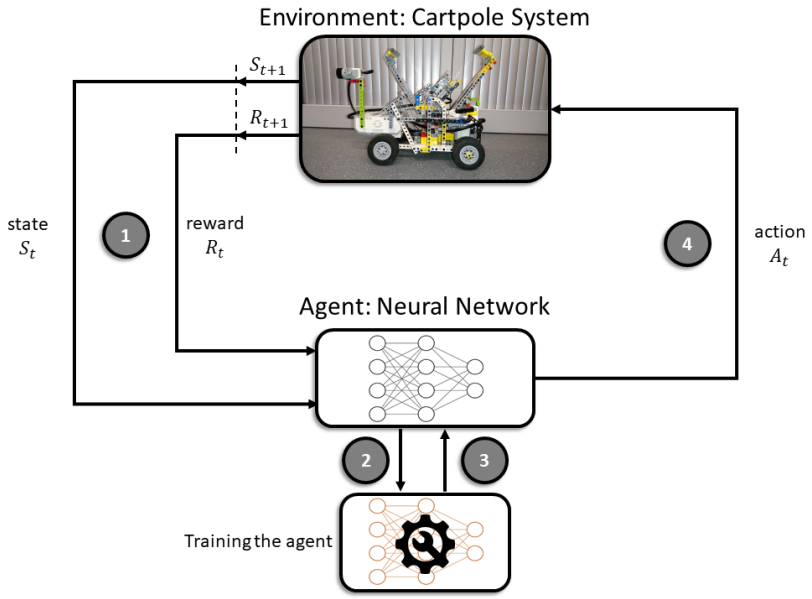


Figure 5.1: The Agent-Environment interaction process in online DQN.

consider delay occurring due to the training methodology used by the DQN algorithm to train the agent. Since the DQN algorithm is an online training approach that interleaves interacting with the environment and learning policy, it is difficult to train the network while interacting with the cartpole system parallelly in a real cartpole system. Training the network at each step would add some amount of delay while the pole is still in motion. This would result in applying the next action at a point when the pole is already out of control.

The main cause of delay in interacting with the real cartpole system is the fact that we perform training while interacting with the system. This is termed as 'Online training'. One possible solution to this was to convert online training to 'Offline training'. In this approach, we repeatedly loaded the network on the EV3 brick, produced control actions observing the state of the cartpole system using the network for a set of episodes, and stored the transactions in a file. The interaction could be then paused and the stored transactions could be sent to the PC where the network would be trained. This effectively demonstrates the offline training approach. Lastly, we can reload the partially trained network from PC on the EV3 and resume the interaction to store the next set of transactions. This process was repeated until we obtain a sufficiently trained network that is capable of balancing the pole completely. A sequence diagram is shown in figure 5.2 that describes the process of offline training of the DQN agent.

Step 1: Obtain a naive online DQN agent using simulation

The main idea was to obtain a configuration of the DQN agent that is capable of balancing the cartpole system. Initially, we started with an online DQN agent with a configuration of [4, 24, 24, 2][1] i.e. 4 neurons in the input layer, 24 neurons in the first hidden layer, 24 neurons in the second hidden layer, and 2 neurons corresponding to two actions: front (or forward) and back (or backward) in the output layer. We used ReLU activation in the hidden layer and linear activation in the output layer. The learning rate was set to 0.001 and Adam optimizer was selected. Since we had built our cartpole system on the hardware, we did not have the mathematical model of

[1]This annotation will be used hereon to describe the structure of the network. The first and last numbers denote the neurons in the input and output layer respectively while the remaining denote the neurons in the hidden layer(s)

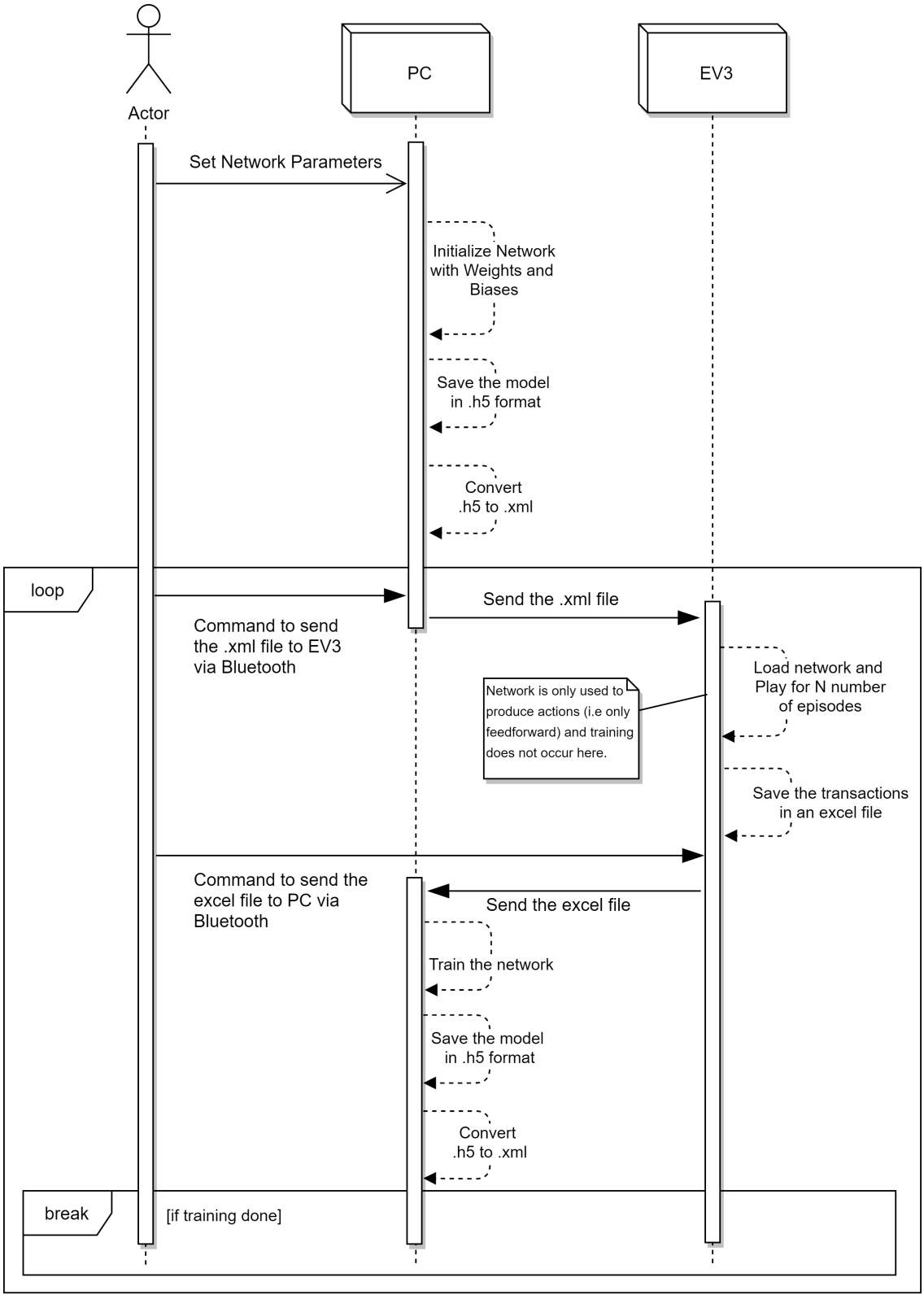
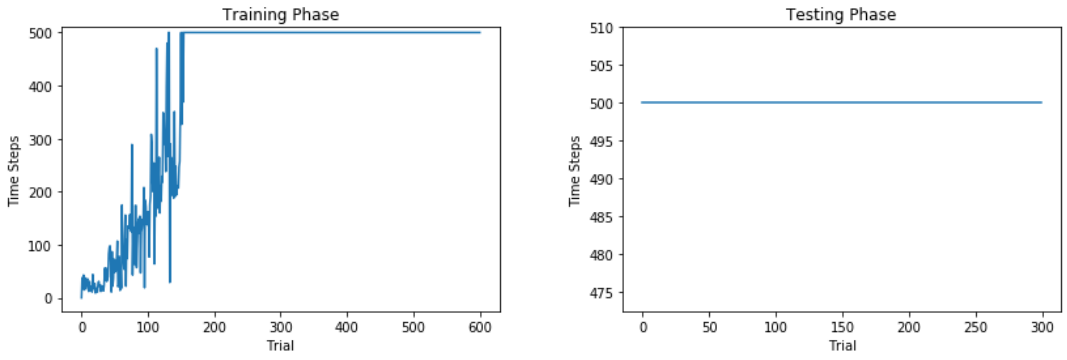


Figure 5.2: Sequence diagram for offline training of DQN agent.

the system. However, there are standardized toolkits in the ML domain that provide an interface for a cartpole balancing problem on simulation. OpenAI Gym is one such toolkit that provides an environment 'Cartpole-v1' that can be used to train and test RL algorithms. Therefore, we trained the DQN agent on simulations using Keras and used the simulation script of OpenAI Gym's Cartpole-v1 environment[2]. The Cartpole-v1 environment's simulation script was changed to have the cartpole parameters such as cart weight, pole weight, pole height, cart weight, etc. equal to the real cartpole system (see table 4.4). The sampling time τ is kept unchanged and is 20 ms as per the simulation script used by OpenAI Gym. The network was able to balance the cartpole and the training and testing results are as shown in figure 5.3. Note that the maximum achievable score as per the simulation script is 500.



(a) Training online DQN agent

(b) Testing online DQN agent

Figure 5.3: Simulation results for training and testing online DQN agent [4, 24, 24, 2]

Step 2: Reduce the network structure

Next, we experimented with the architecture of the agent by reducing neurons and layers and observed that the online DQN agent could not be trained with less than two hidden layers. We found the best configuration of [4, 24, 16, 2] that could successfully balance the cartpole with hardware parameters. The simulation results of the online DQN agent [4, 24, 16, 2] are shown in figure 5.4.

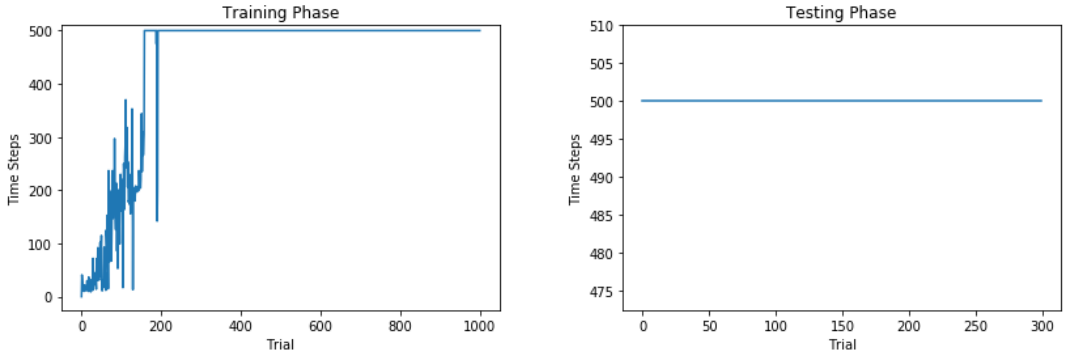
Step 3: Verify offline DQN concept through simulations

Before we attempted to implement offline DQN on the real cartpole system, we tried to imitate the offline training in simulation. Since we know that the online DQN mechanism trains the agent every other step, we simulated offline DQN by skipping a certain number of steps before training the DQN agent [4, 24, 16, 2]. We performed experiments for training operation:

- every step with a chance of 25%,
- after every 5 steps
- after every 10 steps
- after every N steps; where N = [5, 20, 30, 40, 50] and it increases after every 100 episodes.

For every experiment, the trained network was tested on hardware to check how it performs on a real cartpole system.

[2]The simulation script can be found at https://github.com/openai/gym/blob/master/gym/envs/classic_control/cartpole.py



(a) Training online DQN agent

(b) Testing online DQN agent

Figure 5.4: Simulation results for training and testing online DQN agent [4, 24, 16, 2]

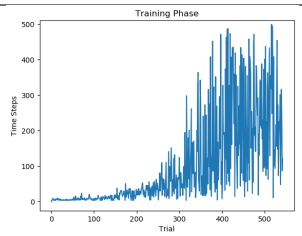
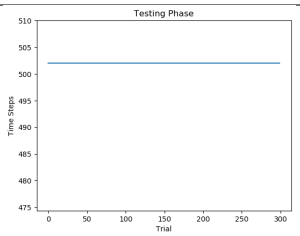
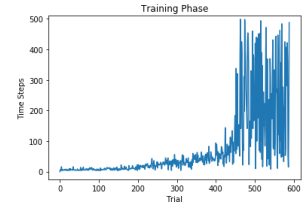
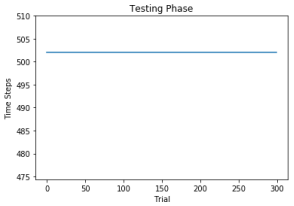
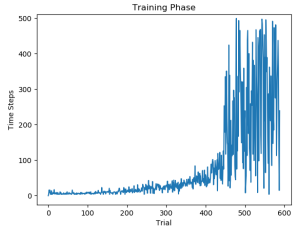
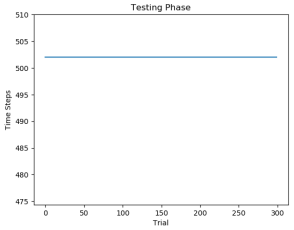
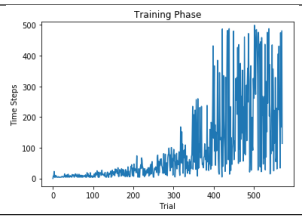
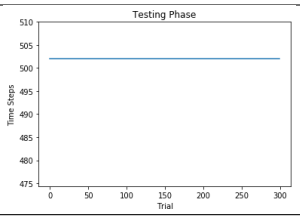
Update Q-network at	**Training**	**Testing**	**Hardware implementation**
Every step with a chance of 25%			P_{StoH}= 17 (∼1.86 sec)
Every 5th step			P_{StoH}= 8 (∼1.03 sec)
Every 10th step			P_{StoH}= 16 (∼1.80 sec)
Every Nth step			P_{StoH}= 18 (∼1.99 sec)

Table 5.1: Simulating offline training

As seen from table 5.1, although the network did not seem to maintain a high score of 500 during the training phase, the network was able to balance the cartpole based upon the results we obtained from the simulation tests. The test was performed for 300 episodes played by the agent. The tests proved that all the trained networks produced from different experiments were

able to perform well and balance the pole until 500 time steps. The reasons for the agents to not to achieve constant high scores in the training phase can be the following:

1. The simulation contains the parameters that match the realistic system that makes the system more complex,

2. In the latter half part of the training, we have an exploration rate of 0.5% which implies that the agent can take a different step with a probability of 0.5%. Since we have made the simulation more complex by inducing realistic parameters and owing to the nature of the problem, the exploration rate implies taking a 'wrong step' with a 0.5% probability.

As a result, we cannot see a constant high score while training the network. The last column in the table 5.1 indicate the numbers of steps/actions taken by the offline DQN agents trained on simulation before the pole fell. We refer to this term as P_{StoH} implying the performance of moving directly from simulation to hardware. The corresponding balance time (in seconds) for steps are also described. We could see that the performance is poor when we use a trained agent from simulation directly on hardware. The primary reason for this effect is called the Reality Gap as described in the section below.

5.3 Reality gap

Simulations are based on mathematical models i.e. set of equations describing the dynamics of a physical system. In our case, we used the Cartpole-v1 environment from OpenAI Gym which contains the dynamics of a real cartpole system in its simulation script. No matter how realistic the simulator is, there always exists a significant gap between simulation and reality and this is termed as the 'Reality Gap'. The proof that there exists a Reality Gap in our case is from the fact that the results from the simulation do not match the realistic behavior. To comprehend the difference, we first introduce some of the control theory concepts.

5.3.1 Control application tasks

We used the control logic provided by the DQN agent to control the real cartpole system. From figure 5.5 we demonstrate how simulation differs from the reality in temporal aspects for our cartpole system. There are three main components in a control application which are described below:

1. Sensing time τ_s : This indicates the time required to get relevant information from the environment. In the simulation world, τ_s is the time required to get the observations from the simulated cartpole environment. In a realistic world, τ_s depicts the time required to measure the distance traveled by the cart, velocity of the cart, angle of the pole from vertical and its angular velocity, and sending the measured values to the controller (EV3 brick).

2. Computation time τ_c: It is the time required by the control logic to produce the control action. In both, simulation and reality, computation time is the time required by the neural network to compute the action with respect to the observed state.

3. Actuation time τ_a: Actuation time corresponds to the time required to execute the task on the system. This holds the same meaning in simulation while in reality, it implies the total of time required to send the action from the controller to the motors and the time required by the motors to execute the command.

As seen from figure 5.5, we have shown how the concept of a sampling period of 110 ms works differently in simulations and on the hardware. The key difference between simulation and practical execution is that, in simulation, we perform all the control tasks (τ_s, τ_c, and τ_a) while pausing the simulation at that point. Once we have the action calculated by the agent we send it to the

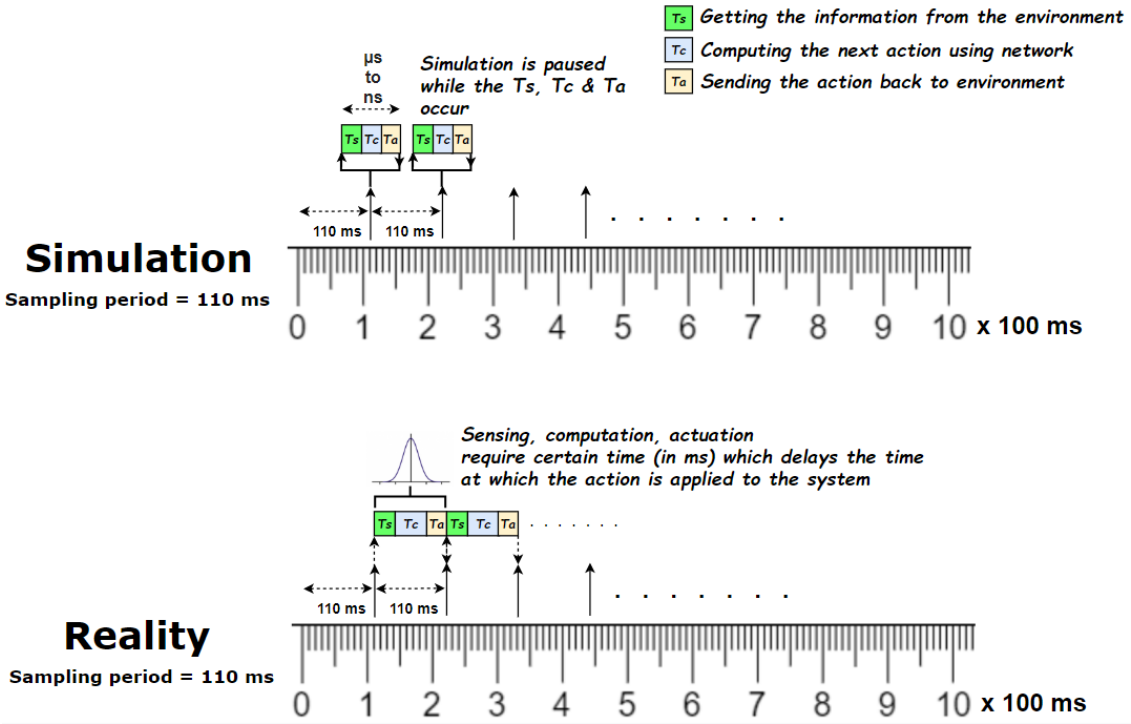


Figure 5.5: Simulating the control algorithm vs. the reality.

system. At this point, the simulation resumes by calculating the next state after 110 ms and returns it to the agent. This process is repeated (see figure 5.5(top section))

Contrastingly, in the realistic world, the point at which the sensing is done (τ_s) and the point at which the action is applied to the system (τ_a) are different. This implies that the network reacts to the cartpole system by providing an action to a state observed at an earlier point in time as the pole is in motion while the τ_s, τ_c and τ_a were being executed. This would result in a delayed action provided to the cartpole (see figure 5.5(bottom section)). Hence, the network trained on the simulation would not be able to balance the pole on the real cartpole system.

Figure 5.6 gives a zoomed-in view of how the action given by the agent is delayed in reality. We observe that the control tasks τ_s, τ_c and τ_a have a varying execution time and are denoted by the mean and variance. Sensing, computation, and actuation tasks have a mean of 15 ms, 60 ms, and 15 ms respectively, and variance of 0.23 ms, 0.3 ms, and 0.25 ms respectively. The neural network shares the maximum time for computation as compared to sensing and actuation tasks. Owing to the inherent sequential nature of the problem, we can observe the next state from the cartpole system only when all the control tasks are completed. This essentially implies the sampling period of the system. As we observed an average of 90 ms for the control tasks to complete, we set the sampling period for experiments slightly higher (i.e 110 ms) on a safer side.

Due to the existence of the reality-gap, the offline DQN agent trained on simulations was unable to balance the pole on a real system. Next, we implement offline training on hardware starting with an untrained agent.

Step 4: Implementing offline training using PyBrain on hardware and Keras on PC

To implement the offline training of the DQN agent, we followed the process described by the sequence diagram in figure 5.2. The training was done on PC using Keras after completion of

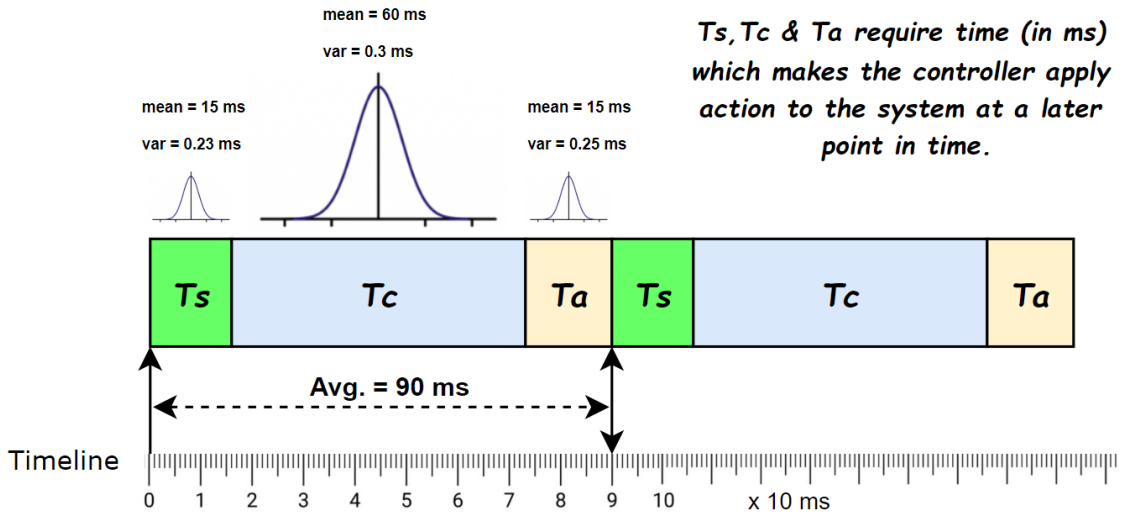


Figure 5.6: Delays encountered for the control tasks in practice.

each episode on the EV3. A single loop of training on the PC followed by running one episode on EV3 took a wall clock time of around 1 minute and 30 seconds. After the completion of 500 episodes i.e. 12.5 hours of training time, the DQN agent could balance the pole for an average of 1.8 seconds with the best time of 2 seconds.

The key aspect behind the success of the DQN agent to be able to balance the pole on simulations is that the agent produces an action based on the current state of the environment i.e. cartpole system. However, this would not hold true in the practical system as the cartpole system will change its state inherently between the time at which its state was sampled, fed to the network, action calculation by the network and the action being sent back to the cartpole system; due to the real-time setting of the problem. To resolve this, we had the following approaches:

1. **Learning the delay**: Predicting the state at which the action will be sent to the cartpole system was the underlying motive behind 'Learning the delay' concept. This approach essentially used two different neural networks: the first neural network predicted the future state (s_{t+1}) based on a temporal sequence of previous states (s_{t-2}, s_{t-1}, and s_t) of the cartpole system; and the second network was the DQN agent that was responsible to calculate the action based on the future state provided by the first network. Thus, the DQN agent would predict the action a_{t+1} for the state s_{t+1} thereby mimicking the key aspect used by a DQN agent in being successfully able to balance the pole in simulations. The architecture of the neural networks used in this approach is as shown in figure 5.7.

   First, we logged a sequence states observed on the hardware when random actions are given to the cartpole system. Next, a set of four consecutive states namely s_{t-2}, s_{t-1}, s_t, and s_{t+1} were grouped per row. Thus, we produced a training dataset as shown in figure 5.8 in which the last column describing state s_{t+1} was considered as the output values and the first three columns describing states s_{t-2}, s_{t-1}, and s_t were considered as the input values. Using the input-output as the training set, we trained the forward predicting network (figure 5.7(left)) having a configuration of [12, 12, 16, 4][3] on PC using supervised learning technique. Once this network was trained, it was taken to hardware where its output was fed to the offline DQN agent to predict the action (front or back).

[3]This was the best configuration obtained from experiments that provided minimum root mean square error value.

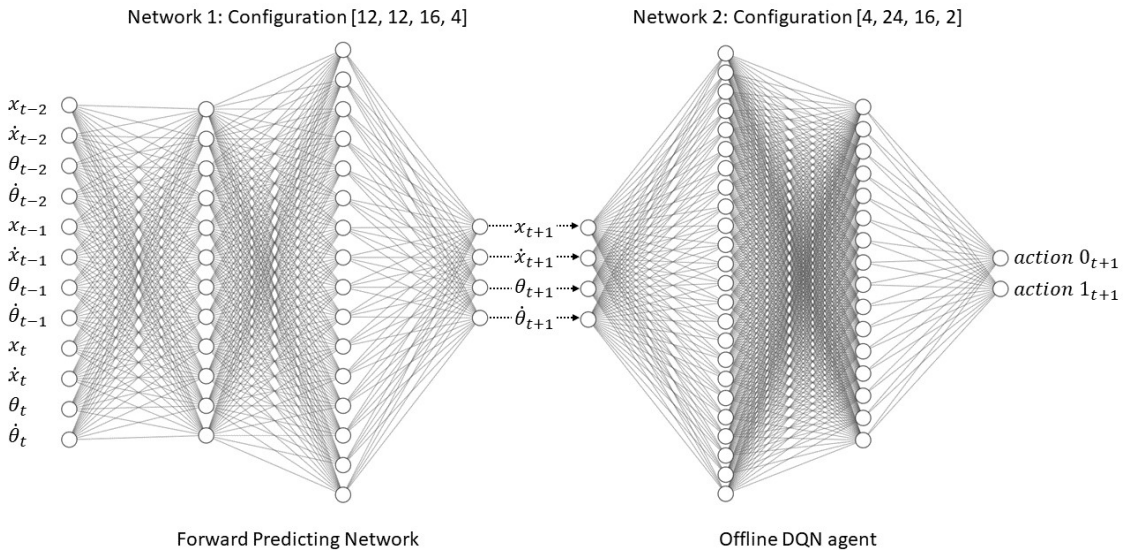


Figure 5.7: Architecture of networks used in the approach of learning the delay.

Input												Output			
State(t-2)				State(t-1)				State(t)				State(t+1)			
x_{t-2}	$\dot{x}_{t-2}$	θ_{t-2}	$\dot{\theta}_{t-2}$	x_{t-1}	$\dot{x}_{t-1}$	θ_{t-1}	$\dot{\theta}_{t-1}$	x_t	$\dot{x}_t$	θ_t	$\dot{\theta}_t$	x_{t+1}	$\dot{x}_{t+1}$	θ_{t+1}	$\dot{\theta}_{t+1}$
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...

Figure 5.8: Dataset for training the forward prediction network.

We performed 500 episodes of offline training corresponding to 14 hrs of wall-clock training time on the hardware using the above approach. The results showed that the agent performed even worse than the previous results where its average balance time was 2 seconds. One of the key reasons for the failure of this approach was that the sampling time was doubled (215 ms) due to the use of two neural networks in calculating the action for the pole balancing task.

2. **Counteracting the delay**: This approach aims to reduce the sampling time caused due to use of two neural networks in the approach of 'Learning the delay'. Despite the forward prediction network was able to approximately predict the future state s_{t+1}, it produced an additional delay which led to the failure of the control over the cartpole system. Therefore, we found a way around this problem by including a simulation script of Cartpole-v1 from OpenAI Gym in the hardware implementation. The parameters of the cartpole used in the simulation script were substituted with the real cartpole's physical parameters to make the predictions as realistic as possible.

   The idea behind 'Counteracting the delay' is similar to that of learning the delay except that in this case, the cartpole's mathematical model i.e. equations of motion describing the dynamics of a cartpole system are used in the hardware implementation. However, in the earlier case, the delay was 'learned' using the interactions with the real cartpole system which makes it a pure reinforcement learning task. By including the equations of motion in the hardware implementation, we predict the future state at which the action will be sent

to the cartpole system. The simulation script contains the sampling time of the hardware using which next state s_{t+1} is calculated based upon the current state s_t. This approach substitutes the forward prediction network from figure 5.7 with a set of equations thereby reducing the sampling time to 110 ms. Figure 5.9 shows how the forward prediction network is replaced by a forward prediction script. The script performs a much fewer number of mathematical operations as compared to the forward prediction network which led to a substantial decrease in overall sampling time of the cartpole system.

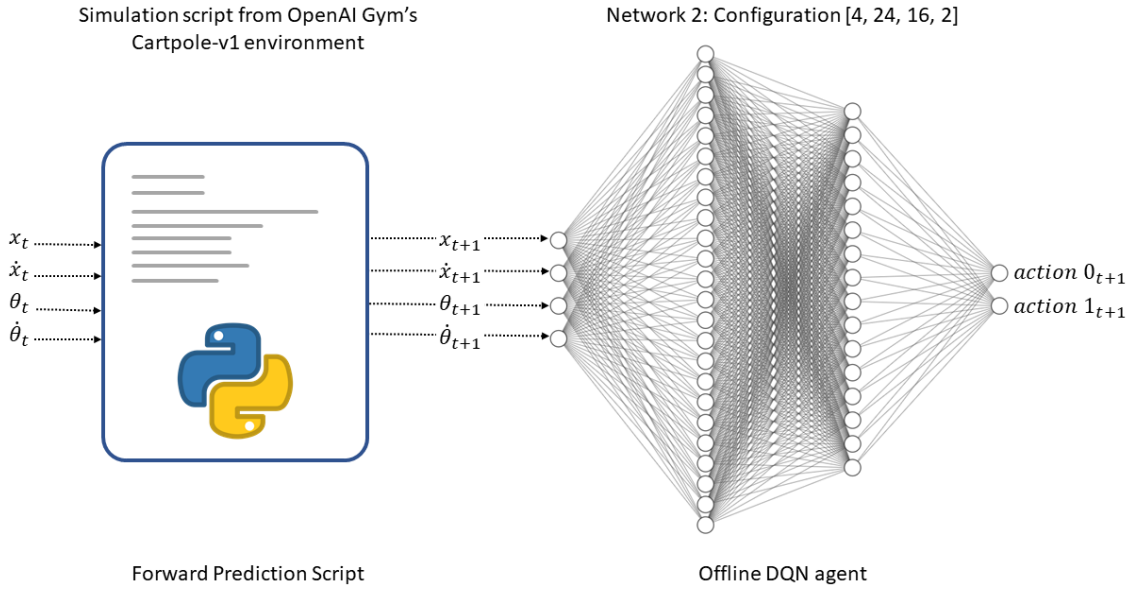


Figure 5.9: Simulation script (left) used along with DQN agent(right) to counteract the delay.

We performed experiments for 1000 episodes i.e. a wall-clock training time of 25 hours using the above mechanism. During the training, the offline DQN agent showed signs of being able to balance the pole for a longer time, with a best of 3.7 seconds of balance time. However, the average balance time was still observed at 2.20 seconds.

After experimenting with all the above approaches and methods to overcome delays, the offline DQN agent was unable to balance the cartpole problem for more than 2.2 seconds on an average. The only possible issue at this point could be the sampling time of 90 ms used on the hardware. Since this value was not explicitly set but was inherently present due to the control tasks (τ_s, τ_c, and τ_a), we attempted to reduce this sampling time so that the DQN agent can provide quicker actions before the pole could fall.

Step 5: Replacing Pybrain by simple NumPy matrix calculations

The sampling time observed on the EV3 brick to sense (i.e. reading the ultrasonic and gyro sensor), compute (feedforward the DQN agent), and actuate (to send the action to the EV3 motors) took an average time of 90 ms. As seen from the leftmost pie chart in figure 5.10, a breakdown of the total sampling time indicates that the neural network consumed the maximum amount of time (67% of the total sampling time) as compared to actuation and sensing tasks. We, therefore, worked on options to reduce the computation time to achieve better sampling time.

Neural networks, under the hood, perform simple matrix multiplications to produce output. An input vector represented as a 1D array is multiplied by a 2D weight matrix followed by adding a 1D bias matrix. The output is finally passed through an activation function that results in a 1D array that is fed to the next layer in the neural network. Thus, in each layer, we perform the

same set of operations until the output of the last layer is obtained. We used this concept and represented our DQN agent i.e. neural network as a sequence of matrix multiplications to produce the action at the output. In general, the output of a neuron with activation function $f(x)$ has an output of the form:

$$y_n = f(\sum_{i=0}^{n}(x_i \times w_i) + b_n) \tag{5.1}$$

where x_i, w_i, and b_i correspond to the inputs, corresponding weights and biases respectively. y_n represents the output of the n^{th} neuron.

We used the NumPy library to perform the matrix multiplications. Since the NumPy library itself was enough to act as the neural network i.e. DQN agent, we could eliminate PyBrain from the hardware implementation. After having successfully substituted the PyBrain with NumPy matrix multiplications as the neural network, we had achieved a 50% reduction in sampling time. As seen from the central pie chart in figure 5.10, the feedforward process of the neural network took around 13 ms out of 43 ms of total sampling time. At this point, we had a balanced proportion of time taken by sensing, computation, and actuation tasks.

Python is an interpreter and one has plenty of scopes to make a python script run faster. We used different profilers to pinpoint sections of code that took a larger amount of time to execute. Python's built-in functions like 'map()' are implemented in C that could run faster than function calls. Using such code optimization techniques, we could further reduce the sampling time to 28 ms in which sensing, computation, and actuation tasks accounted for a contribution of 28%, 43%, and 29% respectively. This corresponds to the pie chart shown to the right in figure 5.10.

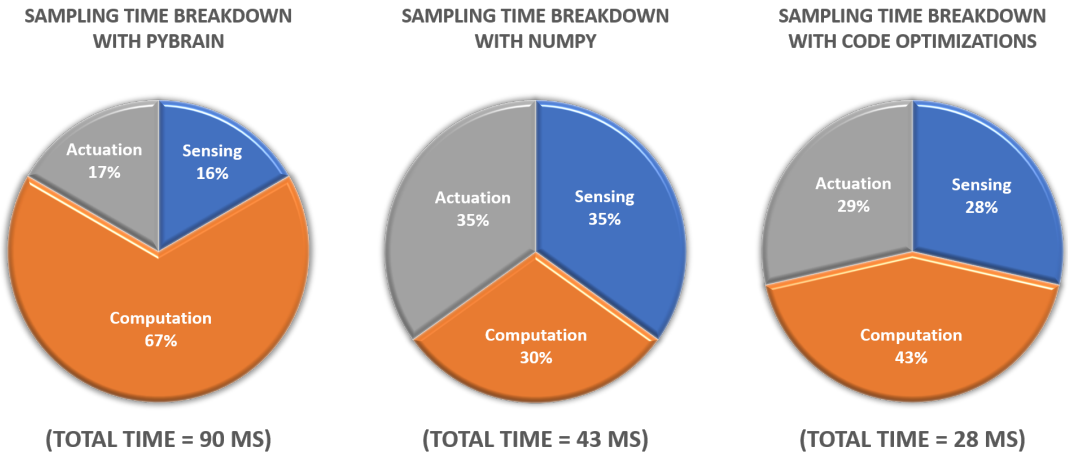


Figure 5.10: Comparison of sampling time using Pybrain (left), NumPy matrix calculations (middle), and NumPy with code optimizations (right) .

In summary, we had achieved optimization of around 70% in sampling time by substituting PyBrain with NumPy matrix multiplications coupled with other code optimization techniques.

Step 6: Retraining with optimized agent

After several changes and optimizations, we had the current offline DQN agent with the configuration [4, 24, 16, 2] that had a sampling time of 28 ms on EV3 using NumPy. We trained this agent on hardware for 500 episodes and observed that after a certain point, the cartpole was frequently moving backward than forward. In other words, the offline DQN agent was predicting the action

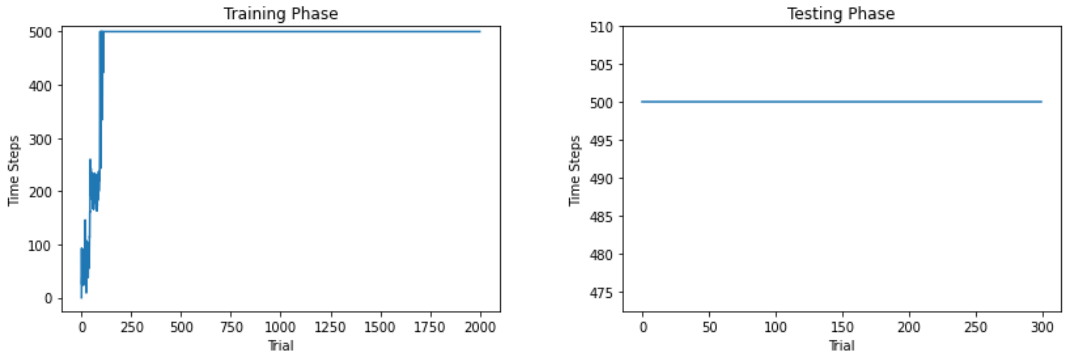
corresponding to moving the cart backward as compared to moving forward. Due to this behavior, the pole was falling faster than expected resulting in a balanced time of 1 second.

On investigating, we observed that the replay memory had stored a significantly uneven number of samples of backward and forward actions. As a result, the DQN agent was trained with higher samples containing backward actions than that of forward actions. This issue occurred due to the design of the cartpole and the inherent nature of the pole balancing problem. As described in the chapter 4 section 4.2, we had built an automated mechanism to reset the pole to an upright position using a set of grabbers. A pole fine-tuning mechanism was used that helped the pole to be held almost vertical allowing the slow-moving grabbers to release the pole. The start of an episode was indicated by the release of the pole by the fine-tuning mechanism. However, the pole could not be held exactly upright as it is at its unstable equilibrium point and the fine-tuning mechanism added a very small bias to the pole to fall back. This resulted in the pole falling backward more often than normal.

Step 7: Uniform sampling from replay buffer

A potential solution to the problem occurring in the above step was to force the DQN agent to sample an equal number of transitions corresponding to backward and forward actions for training itself. This would result in training the agent in equal proportion for forward and backward actions. We initially implemented this on simulations to check if the approach could successfully balance the pole. The simulation results were successful as seen from figure 5.11 and the agent was still able to balance the pole. We also experimented the training by setting exploration = 0 i.e. no exploration of state space and concluded that the agent could still be able to balance the pole without performing any exploration as well.

We used the concept of uniform sampling on hardware and performed 800 episodes of training corresponding to a wall-clock time of 13 hours. However, the agent could not perform better using this approach and we obtained an average balance time of 2 seconds.



(a) Training offline DQN agent

(b) Testing offline DQN agent

Figure 5.11: Simulation results for training and testing offline DQN agent [4, 24, 16, 2] using uniform sampling from replay buffer.

5.4 Results

Following is the summary of the results of using DQN to balance a real-life cartpole system.

1. An online DQN agent [4, 24, 16, 2] was successfully able to balance the pole in simulation.

However, the simulation is quite far from a real-life cartpole system as it disregards various factors such as delays in computation, physical parameters such as friction, etc.

2. The offline DQN agent [4, 24, 16, 2] was also successfully able to balance the cartpole in simulations for the maximum allowable amount of time. However, a direct transition of a trained network from simulation to hardware failed due to the reality-gap.

3. An offline DQN agent was able to balance the pole for an average of 1.8 seconds and the best time of 2 seconds on the EV3 using PyBrain with a training time of 12.5 hours.

4. The offline DQN agent [4, 24, 16, 2] coupled with a forward predicting network failed to balance the pole due to a higher sampling rate caused by the inclusion of a second neural network to enable forward prediction.

5. The best performance of the offline DQN agent [4, 24, 16, 2] was obtained when it was combined with a forward prediction script that could balance the pole for an average of 2.2 seconds and the best time of 3.7 seconds. The major downside in this approach was that it was not a completely model-free approach as it used a mathematical model of cartpole system to predict the future state.

6. An offline DQN agent [4, 24, 16, 2] with a sampling time as small as 28 ms was obtained when the neural network was built as a sequence of matrix multiplications as opposed to using sophisticated machine learning APIs such as PyBrain on EV3.

7. A modified version of sampling data, termed as 'Uniform Sampling' from the replay buffer was used to force the agent to learn both the actions in equal proportion. It was implemented by selecting an equal number of transitions due to both the actions within a batch of training data. However, this did not help in improving the performance of the offline DQN agent [4, 24, 16, 2].

5.5 Conclusion

Using a DQN algorithm to solve a real-life cartpole system designed using EV3 was not successful. The best agent obtained through various experiments and optimizations could provide an average balance time of 2.2 seconds. In a relevant study [27] which aimed at balancing an inverted pendulum, it was proved that even the best performing DRLs were incapable of balancing the inverted pendulum for a long period of time. The best possible balance time achieved by the agents was only 2.5 sec which is even lower than the best score achieved by our offline DQN agent i.e 3.7 seconds. The mentioned study involved a long-lasting training time of 7 days despite which the agents were incapable of balancing the inverted pendulum.

In conclusion, we aim for other reinforcement learning algorithms in order to achieve a better performance in balancing the cartpole problem as the experiments performed using the DQN agent proved the DQN algorithm incapable of being able to solve a real-life pole balancing problem.

Chapter 6

Implementation: NFQ

This chapter discusses the motivation behind opting for the NFQ algorithm after DQN. It entails the process of implementing the NFQ controller from scratch i.e. designing the code, testing it on simulations, and finally implementing it on the hardware. A key takeaway from the chapter is the trick we used to calculate multiple q-values in a single feedforward mechanism from a network designed to produce a single q-value per feedforward path. This move helped reduce the sampling time drastically. We further discuss the existence of state space drift that was observed which could have been a possible hindrance in the performance of the NFQ controller. We perform experiments to overcome this issue. On the whole, the chapter provides details on various configurations of NFQ agents that were experimented until the final controller was obtained that had an average balance time of 2.2 seconds and the best balance time of 3.6 seconds.

6.1 Motivation

The DQN algorithm is popularly attributed to being a slow learning mechanism. In other words, it requires a higher amount of training time with the environment in order to achieve noteworthy performance. This restricts its use for a practical system owing to the limited training time available. A suitable alternative for the DQN algorithm to execute control over a reinforcement learning problem of cartpole balancing in a real-world setting is Neural Fitted Q Iteration or NFQ proposed by Riedmiller [21]. It is known for its sample efficient yet effective training of a neural network to produce high-quality control policies. Primarily, the paper [21] proves the efficacy of the NFQ algorithm by implementing the cartpole regulator benchmark. In the cartpole regulator benchmark, a controller is successful if the pole is still balanced upright by the controller at the end of the episode and the cart is within ± 0.05 m of tolerance at its target position 0. Thus, we aim to implement the same strategy described in the NFQ paper to build an NFQ agent for our real-life cartpole system.

6.2 Experiments

The paper that introduced the NFQ algorithm, described its algorithm and the experiments to prove its success on real-life RL problems. However, it did not publish the source code that was used during the experiments. Therefore, we built our own code in python by using the NFQ algorithm described in the NFQ paper [21].

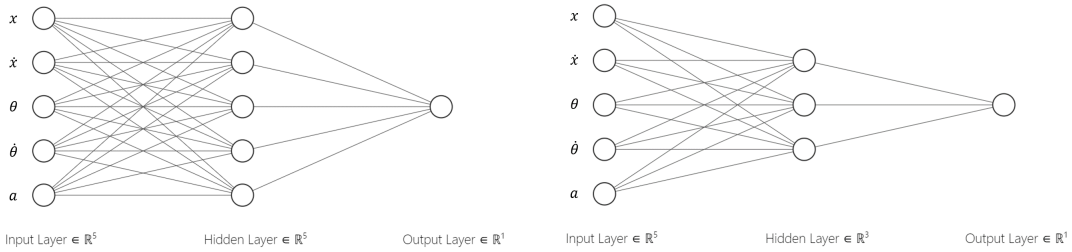
Step 1: Verifying the code for NFQ algorithm on simulations using OpenAI Gym's Cartpole-v1 environment

The NFQ agent that we designed had a configuration [5, 5, 1] with the state variables x, $\dot{x}$, θ, $\dot{\theta}$, and action $a \in [0,1]$ at the input of the network and a single neuron to indicate the q-value of

the state-action pair at the output as shown in figure 6.1a. The settings of the environment i.e. cartpole-v1 from OpenAI Gym were kept unchanged except that the maximum achievable score by an agent was set to 3000 (i.e. 60 seconds, since the sampling time, by default, was 20 ms). The hyperparameters used to train the NFQ agent in the simulation are learning rate: 0.01, activation used in hidden and output layers: Sigmoid, loss: mean squared error, optimizer: rmsprop and epochs: 300. Further, the target value was calculated similar to that of the NFQ paper as,

$$target^l = \begin{cases} 0 & , \quad \text{if } s'^l \in S^+ \\ 1 & , \quad \text{if } s'^l \in S^- \\ 0.001 + min_b Q_k(s'^l, b) & , \quad \text{else} \end{cases} \tag{6.1}$$

with the goal states S^+ being set to cart position is at most 0.1m away from the center AND pole angle is at most ± 0.15 rad i.e. ±8.59°from the vertical. The range of S^- was set to ±0.24m from the center of the track. A pseudo-code to describe the implementation of the NFQ agent on the Cartpole-v1 environment is as given in listing 3.

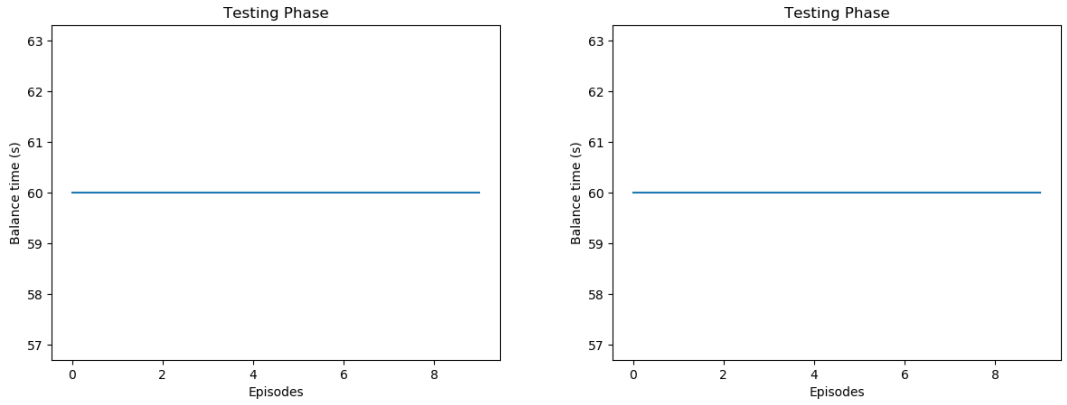


(a) Initial configuration of NFQ Agent. (b) Optimized configuration of NFQ Agent

Figure 6.1: Configuration of NFQ agent used in step 1 (left) and in step 2 (right)

Algorithm 3 Pseudo-code describing implementation of NFQ Agent on Cartpole-v1 env

```
Build the environment 'Cartpole-v1'
Build a network and initialize it with random weights all with sigmoidal activations.
for episode = 1, max episodes do
    env.reset()
    for step = 1, max steps (default:500) do
        Query the NFQ-model and get Q-values for possible actions
        env.step(action)
        action = np.argmin(Q_value)
        if done then
            agent.remember(state, action, next_state)
            Generate lists D_s, D_a, D_s_prime from memory
            Perform one iteration of NFQ algorithm with 300 epochs using Rprop
            Test the current network if it is trained successfully
            if Test successful then
                Exit, as training is completed
            end if
            break
        end if
        agent.remember(state, action, next_state)
        state = next_state
    end for
end for
```



(a) Testing NFQ agent [5, 5, 1] on cartpole-v1 env. (b) Testing NFQ agent [5, 3, 1] on cartpole-v1 env.

Figure 6.2: Test results of NFQ agents in the simulation.

The primary step in implementing the NFQ algorithm was to check the correctness of the code and verify that the translation of the algorithm into a code to incorporate a cartpole balancing problem was done correctly. Therefore, we checked the performance of the NFQ agent on simulation using Keras. We observed that the agent was able to balance the pole for 60 seconds as shown in figure 6.2a after being trained for 127 episodes.

Step 2: Optimizing the structure of NFQ agent [5, 5, 1]

Once we had working code for NFQ and also verified that it works correctly, the next step was to reduce the network structure of the NFQ agent [5, 5, 1] in a way that it can still be able to balance the pole in the cartpole system on simulations. Reducing the size of the NFQ agent could be done in the following ways:

1. Removing the hidden layer completely: We started with completely excluding the hidden layer of 5 neurons from the NFQ agent [5, 5, 1]. However, the NFQ agent did not converge when there was no hidden layer in the network. This was obvious since a hidden layer is responsible for incorporating the non-linearity in solving a dynamic, non-linear cartpole balancing problem.

2. Reducing the number of neurons from the hidden layer: Upon experimentation, we obtained an NFQ agent with the structure [5, 3, 1] as shown in figure 6.1b, that was also capable of balancing the pole in simulation. The result of testing the NFQ agent [5, 3, 1] after training for 87 episodes is as shown in figure 6.2b.

Another scope for improving the network structure was to replace the sigmoidal activation function with a computationally efficient activation function. A sigmoid activation function is given by,

$$\text{Sigmoid Activation function } f(x) = \frac{1}{1 + e^{-x}} \tag{6.2}$$

As seen from equation 6.2, passing a value into a sigmoid function requires calculating an exponent term which is computationally expensive. As a consequence, it would consume additional time for computation on the hardware as sigmoid activation is used for each neuron. This could result in increased overall sampling time. To avoid this phenomenon, we experimented using a Rectified Linear Unit (ReLU) activation function in the hidden layers since it has a minimal impact on computation time. The ReLU activation function is given by,

$$\text{ReLU Activation Function } f(x) = \begin{cases} 0 & , \quad \text{if } x < 0 \\ x & , \quad \text{otherwise} \end{cases}$$

The experiments concluded that the NFQ agent could not converge when a ReLU activation function was used in hidden layers. Therefore, we retained the earlier configuration of the NFQ agent [5, 3, 1] with all sigmoidal activations with hardware parameters on the simulation.

Step 3: Training the NFQ agent [5, 3, 1] with hardware parameters

The results obtained until this point were using the OpenAI Gym's Cartpole-v1 environment with the default parameters. As a next step, we changed these parameters and set the values equal to the parameters of the real cartpole system. We also set the sampling time to 40 ms which we obtained by running the NFQ agent [5, 3, 1] twice (corresponding to producing two actions) on the EV3 brick.

The results proved that the NFQ agent [5, 3, 1] was able to balance the pole after playing 145 episodes in the simulation. This was a key result as it proved that the NFQ algorithm was able to learn the cartpole balancing problem concerning the hardware setup used in our thesis. We tested the agent for 10 episodes with each episode lasting for 120 seconds. The agent was able to balance the pole for the maximum duration in all the testing phases (see figure 6.3).

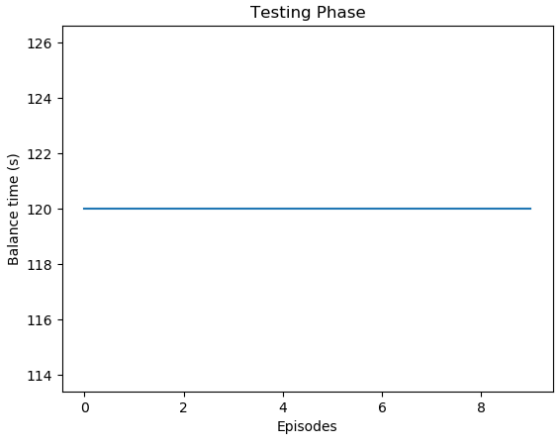


Figure 6.3: Simulation result of testing the NFQ agent [5, 3, 1] with hardware parameters

Step 4: Training NFQ agent [5, 3, 1] on the hardware

With the success of NFQ implementation with two actions on the simulations, we could start implementing NFQ on the hardware. The process that we followed to implement NFQ on EV3 was similar to that of DQN as shown in the sequence diagram 5.2 except that, whenever the network was loaded on the EV3 brick, it was allowed to play for only one episode and then sent back to PC for retraining. We also used the forward prediction script as shown in figure 6.4 so that the time at which the action was sent to the cartpole system would correspond to the state of the cartpole at that time.

The NFQ agent was trained for 1016 episodes on hardware (accounting to 22.5 hrs of training time) which resulted in generating 26,000 training data samples. The scores obtained by the agent while training on the hardware are shown by the graph in figure 6.5. The best balancing time achieved by the NFQ [5, 3, 1] agent was 3.1 seconds after episode 308 (corresponding to the peak in the graph shown in figure 6.5). On average, the agent could balance the pole for 2.15 seconds.

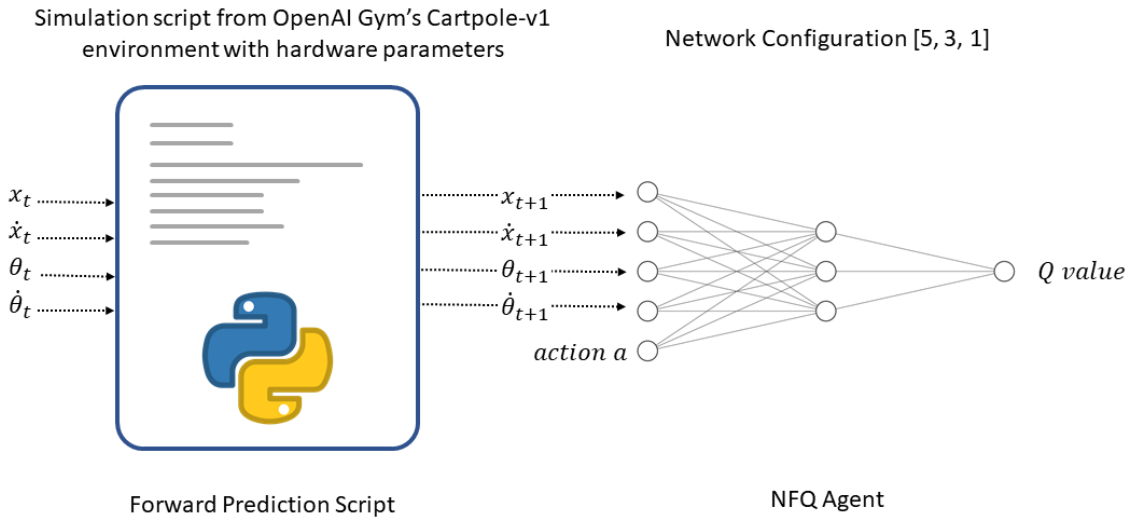


Figure 6.4: Hardware setup for training NFQ agent [5, 3, 1].

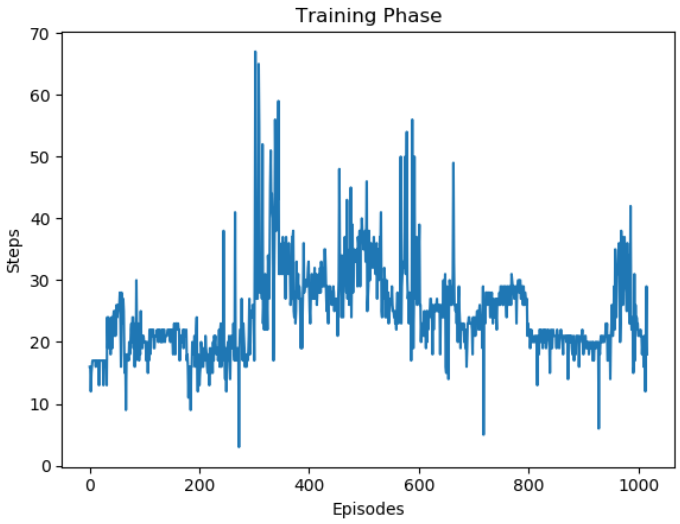


Figure 6.5: Performance of NFQ agent [5, 3, 1] on the hardware

At this point, the EV3 Lego Large motor (6009430) that drove the rear wheels of the cartpole system was damaged due to excessive training sessions carried out on it. It lasted for a massive 100 hours of vigorous use while training different AI controllers on the cartpole system. Since the real wheel axle assembly was complex, most of the parts of the cartpole system had to be dismantled and reassembled to replace the faulty large motor with a new motor.

To improve the performance of the NFQ agent, we tried to reduce the sampling time further to allow the agent to have more frequent interaction with the cartpole system. One possible way was to use a Look-up Table (LUT) for the sigmoid function so that it would require fewer calculations due to the activation function. Thus, we implemented a dictionary comprising of 1000 rows that consisted of input-output the pairs of the possible range of values that could be fed to the sigmoid function during the control of the cartpole problem. Additional logic was also required to be included to select the closest key in case an exact key was not found for a particular input value.

Since the dictionary was created ahead of the start of the control loop, it did not directly interfere with the sampling time. Surprisingly, the results of using a dictionary for a faster calculation of activation function were not as expected. The sampling time rose to 150 ms as compared to the initial 40 ms, which proved that the EV3 took a longer time to search through 1000 rows of the dictionary than to calculate the output of sigmoidal activation in the runtime. Therefore, the idea of using the LUT was discarded.

Another approach that we used to reduce the sampling time of the NFQ agent was by changing its mechanism of calculating action based on the input state. Originally, to choose an action out of possible options (i.e. [0, 1] corresponding to forces [-14N, 14N]) suitable for a given state [x, $\dot{x}$, θ, $\dot{\theta}$], the NFQ agent first feedforwards the input set [x, $\dot{x}$, θ, $\dot{\theta}$, 0] to produce the q-value for action '0'. Next, it again feedforwards the input set [x, $\dot{x}$, θ, $\dot{\theta}$, 1] to produce the q-value for action '1'. Finally, it compares the two q-values and chooses the action corresponding to the minimum q-value. This process requires two iterations (or feedforward mechanisms) of the network. Therefore, we experimented to check if both the q-values could be obtained with a single feedforward of the network.

It was indeed possible to produce two q-values through a single feedforward process by using a 2D array of inputs being fed to the network. Since we used simple matrix multiplications to represent a neural network, using a 2D array in place of a 1D array of input was not too difficult. An example to illustrate the feeding of a 2D matrix of input to the NFQ is as shown in figure 6.6. This trick helped to reduce the overall sampling time such that it took only 22 ms to calculate both the q-values as compared to the earlier time of 40 ms to calculate two q-values. Thus the sampling time was reduced by 45%

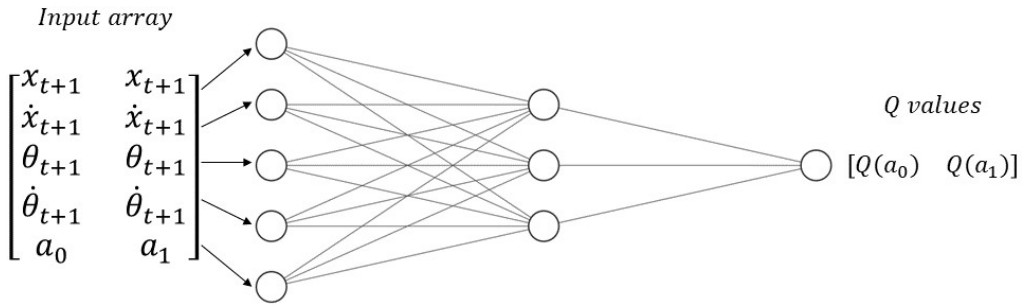
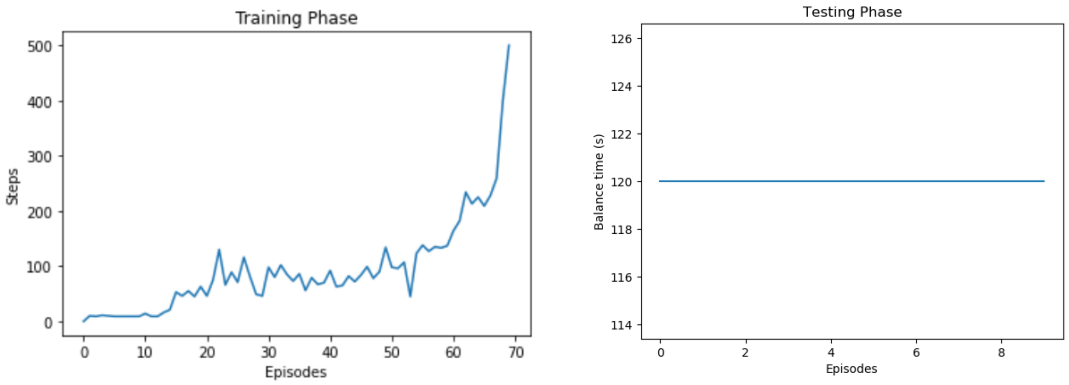


Figure 6.6: Illustration of a single feedforward of NFQ agent [5, 3, 1] to produce multiple q-values.

We took the advantage of this move by adding a neutral action to the action set [0, 1, 2] corresponding to forces [0N, -14N, +14N]. The author of the NFQ paper, suggests in [20], that it is useful to have neutral actions that do not put additional energy into the system. With the additional neutral action and the trick to producing multiple q-values in one go, we obtained an NFQ agent [5, 3, 1] which had a sampling time of 25 ms to choose an action out of [0, -14N and +14N]. Next, as a sanity check, we planned to implement this agent on simulations first, before directly implementing it on the hardware.

Step 5: Implementing NFQ agent [5, 3, 1] with 3 actions: [0N, -14N, +14N] on simulations with hardware parameters.

The training of the NFQ agent lasted for 68 episodes after which the agent was able to balance the pole with a 100% accuracy. Figure 6.7a depicts the learning curve of the NFQ agent during the training phase whereas figure 6.7b describes the performance of the trained NFQ agent. The testing phase comprised of 10 episodes, each lasting for a maximum of 120 seconds (i.e. 4800 steps with a sampling time of 25 ms.)



(a) Training NFQ agent [5, 3, 1] with three actions (b) Testing NFQ agent [5, 3, 1] with three actions

Figure 6.7: Performance of NFQ agent [5, 3, 1] with 3 actions: [0N, -14N, +14N] on the simulations.

Step 6: Implementing NFQ agent [5, 3, 1] with 3 actions: [0N, -14N, +14N] on the hardware

As the 3-action NFQ agent could succeed in balancing the pole on the simulations with hardware parameters, we moved to hardware to train the same agent. After training for 600 episodes (≈12 hrs of wall-clock time), the NFQ agent [5, 3, 1] could reach the best balancing time of 3.6 seconds and an average balance time of 2 seconds. The best score was attained by the agent in the 49^{th} episode.

Despite 3.6 seconds of balance time was the best score we had obtained until this point after experimenting several configurations of DQN as well as NFQ agents, the average performance of 2 seconds was not noteworthy. The paper that introduced NFQ proved its efficiency only through simulations. However, the same author had experimented on a real-life cartpole system in [20] using an NFQ agent. The cartpole setup used in the experiment involved a swing-up and balance task. This was the only key difference in the hardware setup as compared to the cartpole system that we have used in the thesis. Further, the author used an effective method to overcome the delays caused due to sensing, computation, and actuation tasks. The state information was augmented with historical information by including the previous action in it. Thus, we decided to implement the strategy used by the author in [20].

Step 6: Implementing NFQ agent [6, 3, 1] with 3 actions: [-14N, 0N, +14N] on the hardware

We designed the NFQ agent [6, 3, 1] with the tanh activation function (since this was used in the paper [20]) in the hidden layer and sigmoid activation at the output. The network configuration used in the [20] was [6, 20, 20, 1] and due to better hardware setup, the sampling time was 0.01 seconds. Since we used comparatively naive hardware to build the cartpole system, we could not include a higher number of neurons as it would lead to increased sampling time. Further, we also applied the trick of scaling the input values as used in [20]. To normalize the input values to have a mean of 0 and a standard deviation of 1, we used the z-score formula as given in 6.3. Furthermore, to exactly match with the setup from the paper, we also modified the action set to [-14N, 0N, +14N] from [0N, -14N, +14N]. We continued to include the calculation of multiple q-values in one go as done in step 4. With the above configurations, the NFQ agent [6, 3, 1] required 50 ms of sampling time. The increased sampling time was due to the calculation of z-scores for the input array as well as the inclusion of one additional input neuron. Since we had included the past action a' in the state variables, the forward prediction script was not used during implementation. The architecture used for training NFQ agent [6, 3, 1] on the hardware is as shown in figure 6.8.

$$Z = \frac{x - \mu}{\sigma} \tag{6.3}$$

where,
Z = Standard score
x = input value
μ = mean of the sample
σ = standard deviation of the sample

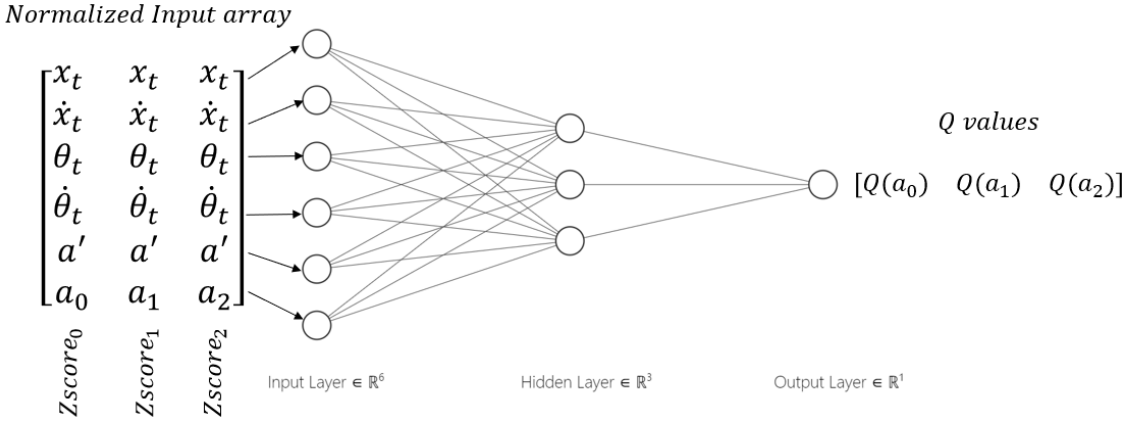


Figure 6.8: Architecture of NFQ agent [6, 3, 1] showing normalized input and no forward prediction script.

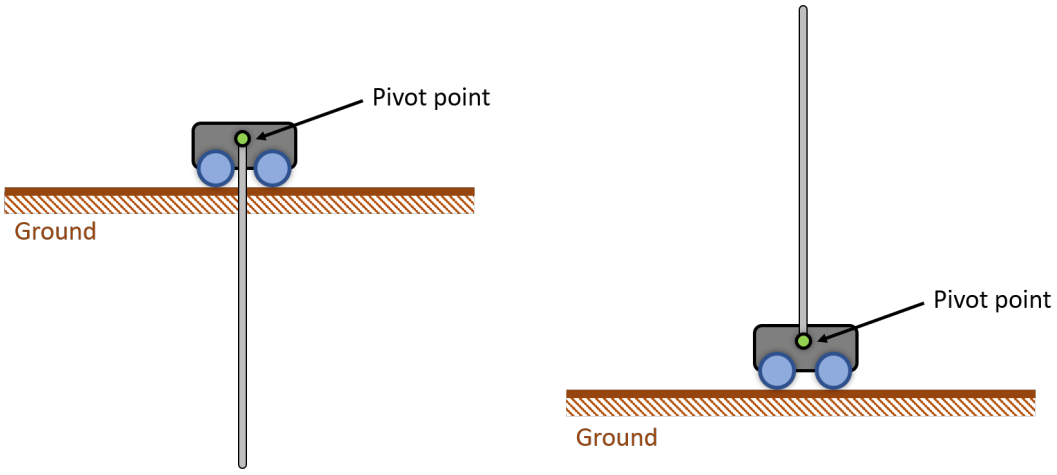
The NFQ agent was trained for 700 episodes i.e. a training time of 15.5 hours. We observed that the agent could balance the pole for a maximum of 2.9 seconds in the 334$^{\text{th}}$ episode, and its average balance time was still around 2.2 seconds. We investigated possible issues for the agent to be unable to achieve a higher average balance time than 2.2 seconds. The cartpole system that we had built had an inherent issue due to the initialization mechanism used in it. As described in the hardware setup (section 4.2), the combination of using a grabber mechanism and a fine-tuning mechanism to keep the pole upright before the start of an episode led to the issue of 'State Space drift'.

State Space Drift

The hardware setup that we built as a demo for representing a cartpole system moved on a set of four Lego wheels as shown in figure 4.1b. The pole balancing task should be initialized at either of the equilibrium points shown by figures 6.9a and 6.9b. Since the cartpole system we built would move on the ground, the pole could not be initialized at the stable equilibrium point (6.9a), and hence, it had to be initialized at the unstable equilibrium point(6.9b).

The fine-tuning mechanism was built to allow the grabbers to release the pole while keeping the pole almost upright. The release of the fine-tuning mechanism indicated the start of an episode. The EV3 Gyro sensor installed near the pivot point was reset every time before the release of the fine-tuning mechanism to avoid the Gyro sensor drift[1]. Using such a setup to reset the pole vertically upright, created a difference in state space when the pole was initialized slightly away from the center at different episodes. This is shown in figure 6.10 where the pole is initialized at three different positions: I_0, I_1, and I_2 for three episodes:0, 1, and 2 respectively. The shaded

[1]This issue is caused inherently due to the manufacturing of the sensor.



(a) Stable equilibrium point of cartpole (b) Unstable equilibrium point of cartpole

Figure 6.9: Stable and unstable equilibrium points in a cartpole system.

cones behind every pole indicate the state space as seen by the controller for each of the episodes. This phenomenon is termed as 'State Space Drift'.

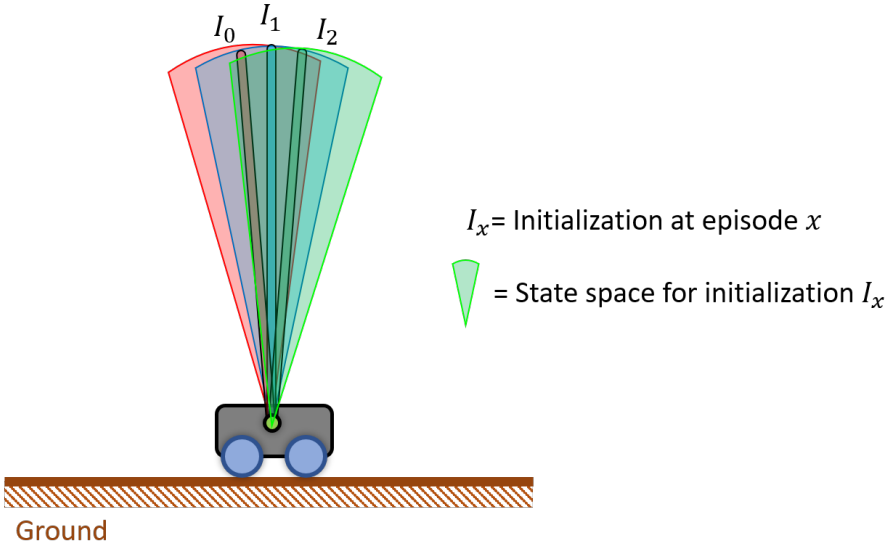


Figure 6.10: The pole being initialized at different starting points leading to a state space drift

Since the hardware could not be upgraded/modified to keep the pole exactly upright, we cannot avoid the presence of state drift in the cartpole system when the pole initialization is done at the unstable equilibrium point. A solution to this was to make the controller more robust to the state space it observes by adding some Gaussian noise to the state space variables x, $\dot{x}$, θ, and $\dot{\theta}$.

Step 7: Retraining the NFQ agent [6, 3, 1] with noise on the hardware

The addition of Gaussian noise to the state variables did not affect the sampling time of the cartpole system and we still could maintain the sampling time of 50 ms (on an average) to train the NFQ agent [6, 3, 1] with noise on the hardware. The agent achieved the best balance time

of 2.79 seconds after training for 500 episodes (≈10 hrs of training time) and the average balance time was still observed around 2 seconds. Therefore, the addition of noise could not contribute to increased performance.

After further investigation, we observed that the z-scores that we calculated to normalize the input values could be a possible issue in not allowing the agent to learn properly. The idea behind calculating the z-scores of the input was to scale and normalize the input variables. The input variables had the following ranges:

- x: -2.4m to 2.4 m
- $\dot{x}$: -∞ to +∞
- θ: -0.6019 rad to +0.6019 rad i.e. -35° to +35°.
- $\dot{\theta}$: -∞ to +∞
- a': [0, 1, 2]
- a: [0, 1, 2]

While calculating the z-score, we had included the current action value a, and the past action value a' as well. Since we feed three sets of z-scores for three different actions as a 2D input array, we were feeding the network with three completely different sets of inputs. To exemplify, let the state space observed be,

$$\begin{bmatrix} x \\ \dot{x} \\ \theta \\ \dot{\theta} \\ a' \end{bmatrix} = \begin{bmatrix} 0.02 \\ 0.5 \\ -0.01 \\ -0.13 \\ 1 \end{bmatrix}$$

It implies that the previous action i.e 1 (≡ 0N), taken by the controller led the cartpole to move into the state given by [x, $\dot{x}$, θ, $\dot{\theta}$]. The z-score of the input is calculated as,

$$\begin{bmatrix} x \\ \dot{x} \\ \theta \\ \dot{\theta} \\ a \end{bmatrix} = \begin{bmatrix} 0.02 \\ 0.5 \\ -0.01 \\ -0.13 \\ 1 \end{bmatrix} = \begin{bmatrix} -0.60796797 \\ 0.53197197 \\ -0.67921421 \\ -0.9641992 \\ 1.71940941 \end{bmatrix}$$

Next, we produced a 2D matrix combining the state variables with each action. The Z-scores are calculated as shown below:

$$\begin{bmatrix} x & x & x \\ \dot{x} & \dot{x} & \dot{x} \\ \theta & \theta & \theta \\ \dot{\theta} & \dot{\theta} & \dot{\theta} \\ a' & a' & a' \\ a_0 & a_1 & a_2 \end{bmatrix} = \begin{bmatrix} 0.02 & 0.02 & 0.02 \\ 0.5 & 0.5 & 0.5 \\ -0.01 & -0.01 & -0.01 \\ -0.13 & -0.13 & -0.13 \\ 1 & 1 & 1 \\ 0 & 1 & 2 \end{bmatrix} = \begin{bmatrix} -0.52775592 & -0.80204399 & -0.72569983 \\ 0.67854333 & 0.22002977 & -0.08459078 \\ -0.60314963 & -0.8659236 & -0.76576914 \\ -0.90472444 & -1.12144204 & -0.9260464 \\ 1.93510506 & 1.28468993 & 0.58323115 \\ -0.57801839 & 1.28468993 & 1.918875 \end{bmatrix}$$

It is evident that from a network's point of view, the input matrix described three completely different states of the cartpole which was wrong. We, therefore, excluded the past action a and current action a which produced the input matrix as,

Episode	Balance Time (in sec)
3	2.67
244	2.25
258	2.31
262	2.19
377	2.42
579	2.43
674	2.66
699	2.35
720	2.57

Table 6.1: NFQ [6-6-1] 3 actions hardware performance

$$\begin{bmatrix} x & x & x \\ \dot{x} & \dot{x} & \dot{x} \\ \theta & \theta & \theta \\ \dot{\theta} & \dot{\theta} & \dot{\theta} \\ a' & a' & a' \\ a_0 & a_1 & a_2 \end{bmatrix} = \begin{bmatrix} 0.02 & 0.02 & 0.02 \\ 0.5 & 0.5 & 0.5 \\ -0.01 & -0.01 & -0.01 \\ -0.13 & -0.13 & -0.13 \\ 1 & 1 & 1 \\ 0 & 1 & 2 \end{bmatrix} = \begin{bmatrix} -0.31189143 & -0.31189143 & -0.31189143 \\ 1.68421373 & 1.68421373 & 1.68421373 \\ -0.436648 & -0.436648 & -0.436648 \\ -0.93567429 & -0.93567429 & -0.93567429 \\ 1 & 1 & 1 \\ 0 & 1 & 2 \end{bmatrix}$$

In the above case, it is clear that the network would calculate q-values for 3 different actions taken in the same state.

An advantage of excluding the actions while standardizing the inputs was that the number of times the z-score had to be calculated dropped by a factor of 3 as we calculated z-score for each state variable-action pair. This reduced the sampling time and created room to add neurons. As a matter of fact, an entire hidden layer could be added to the previous network configuration. We chose to add neurons as that would make the network more capable and powerful to handle the nonlinearity and complex nature of the real-cartpole system.

Step 8: Training the NFQ agent [6, 6, 6, 1] with noise on the hardware

We could fit in a second hidden layer with six neurons and the original hidden layer was added with 3 neurons whilst maintaining the same sampling time of 50 ms owing to the reduction in z-score computation (see figure 6.11). We trained the NFQ agent [6, 6, 6, 1] with added Gaussian noise on the hardware. Table 6.1 enlists the best scores obtained by the NFQ agents in terms of balance time. Despite the additional hidden layer, higher computation power in terms of neurons, the NFQ agent [6, 6, 6, 1] could balance the pole for a maximum of 2.67 seconds and an average balance time of 2.2 seconds after training for 800 episodes (≈18 hrs of training time).

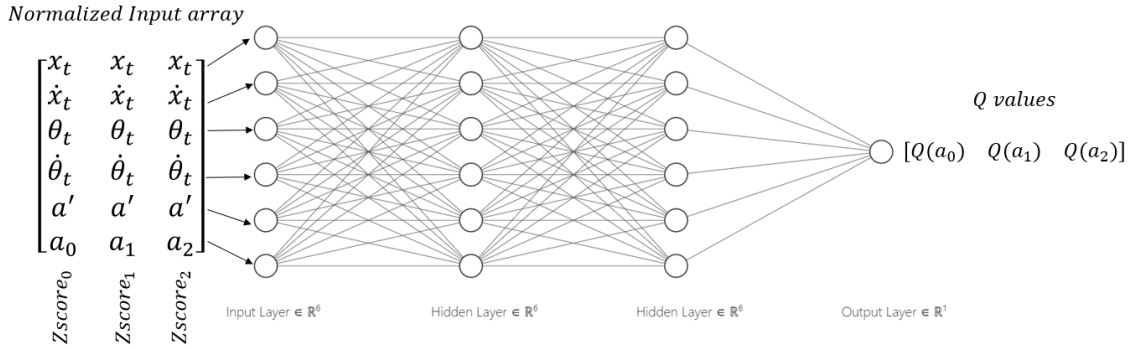


Figure 6.11: Architecture of the NFQ agent [6, 6, 6, 1]

Apart from the above variations in configurations, we have also tried different modifications by changing the choices for X^+, X^-, and addition of artificial states namely [0,0,0,0,0,0] in the spirit of hint-to-goal heuristic [21] as suggested in [20]. Nevertheless, the performance of the agent remained the same.

6.3 Results

The summary of results obtained through implementing NFQ algorithm to control a real-life cartpole problem are noted below:

1. The NFQ agent [5, 5, 1] along with its optimized version of configuration [5, 3, 1] were both able to execute complete control of a simulated cartpole system provided by OpenAI Gym. The agents were able to learn the task of balancing the pole within 127 and 87 episodes respectively.

2. The NFQ agent [5, 3, 1] was also able to balance the cartpole system on the simulation with hardware parameters. It took 145 episodes to learn the task successfully.

3. Accounting to a wall-clock training time of 22.5 hours, the NFQ agent [5, 3, 1] could balance the pole for a maximum of 3.1 seconds on the hardware and an average balance time of 2.15 seconds. The sampling time used was 40 ms.

4. A trick to calculate multiple q-values in one iteration of the network was helped reduce the sampling time significantly. This provided scope to add a neutral action to the action set which is necessary for the context of controlling a real dynamical system. Therefore, a NFQ agent [5, 3, 1] with 3 actions: [0, 1, 2]≡[0N, -14N, +14N] and a sampling time of 25 ms was designed. The experiments showed that it was capable to balance the pole infinitely on the simulation with hardware parameters. It learned the pole balancing task in merely 68 episodes.

5. The NFQ agent [5, 3, 1] could also perform well on the hardware with the use of three actions and a sampling time of 25 ms. It achieved the best balance time of 3.6 seconds and an average of 2 seconds. It was trained for a total of 600 episodes accounting for ≈12 hours of training time.

6. A new NFQ agent [6, 3, 1] coupled with the action set: [0, 1, 2]≡[-14N, 0N, +14N] was designed inspired by [20]. It used normalized inputs and a sampling time of 50 ms on the hardware. After being trained for 700 episodes (≈15.5 hours of training), the agent could balance the pole on the hardware for a maximum of 2.9 seconds and an average of 2.2 seconds.

7. The same NFQ agent [6, 3, 1] was retrained with Gaussian noise to eliminate the state space drift. However, the agent completed 500 episodes (≈10 hrs of training time) to be able to balance the pole for a maximum of 2.79 seconds and an average of 2 seconds.

8. On discovering an improper calculation of z-scores at the input of the NFQ agent [6, 3, 1], a correct method was proposed. The corrective measure allowed the addition of neurons to the network since a few seconds were saved avoiding unnecessary computations during input normalization. This led to a design of the NFQ agent [6, 6, 6, 1] with normalized input and a sampling time of 50 ms. This agent was trained for 800 episodes on the hardware (totaling ≈18 hours of training time) during which it balances the pole for a maximum of 2.67 seconds.

6.4 Conclusion

To sum up, the NFQ agent in general performed better than the DQN agent in terms of executing a model-free control. However, the average balance time of 2.2 seconds was similar to that of DQN agent and not yet significant. We have referenced the NFQ algorithm on account of its success in being able to balance a real-life cartpole system. The author of the NFQ paper has implemented the NFQ algorithm with enough evidence that NFQ is tailor-made to control real RL problems. Despite following a similar procedure taken in [20], the settings in our experiments are not exactly the same in terms of network architecture, sampling time, and the hardware setup. The author has designed a slightly complex swing-up and pole balancing task through sophisticated components accompanied by real-time software. To exemplify, the hardware used in the paper is capable of providing a fixed sampling time of 10 ms which is very small. On the other hand, the demo product that we built for the thesis using the LEGO Educational kit does not guarantee a fixed or less variable response time. The EV3 brick uses embedded Linux at its heart powered with a comparatively simpler ARM9 core. The problem becomes worse when the hardware has very limited cache memory and slower clock. As a result, it becomes nearly impossible to balance a real-time dynamical control problem like the cartpole system using a LEGO educational kit. Nevertheless, we have a few agents that can manage to balance the pole for an average of 2.2 seconds running on top of non-real-time hardware.

Chapter 7

Implementation: GA

In this chapter, we describe the motivation for using the Genetic Algorithm in our thesis. We elaborate on how the experiments were carried out to optimize the existing solutions of the NFQ agent. The key insight from this chapter is that the GA was able to produce a better quality controller when it was used alone to produce the controllers. In other words, the GA was ineffective when it comes to optimizing existing solutions whereas it excelled when it was provided with completely untrained agents. The effective way of generating best-performing agents through GA was by training the agents initially on the simulation and further continuing their training on the hardware. Therefore, a best performing GA agent is produced at the end of this chapter that could balance the pole on the hardware for an average of 2.42 seconds and the best of 6.03 seconds. The GA agent was named after its index in the population as Agent-45.

7.1 Motivation

In the world of neural networks, the term 'parameters' is often referred to as the weights and biases of the network. When we train a network, it essentially learns the parameters until it is able to solve the task given to it. The better the set of parameters learned, the better is the performance of the network on the task. Therefore, poor performance can be attributed to a sub-optimal set of parameters of the network. We use Evolutionary Algorithms to optimize and improve sub-optimal agents obtained in this thesis. Genetic Algorithm (GA) is a popular classical evolutionary algorithm. In principle, it is based on Charles Darwin's theory of natural evolution "survival of the fittest" and uses a gradual and random tuning of the network parameters until the network's performance increases to an expected level. Based on this concept, we chose to use a genetic algorithm to optimize the existing best performing AI agents and improve their performance to some extent. In our case, we had obtained the following set of agents that had comparable performance:

1. An offline DQN agent [4, 24, 16, 2] with a sampling time of 90 ms, best balancing time of 3.7 seconds, and an average balance time of 2.2 seconds. It used a forward prediction script to overcome the delays encountered on the hardware.

2. An NFQ agent [5, 3, 1] with a sampling time of 40 ms, best balancing time of 3.1 seconds, and an average balance time of 2.15 seconds. It used a set of two actions: [-14N, +14N] and a forward prediction script to execute the control.

3. An NFQ agent [5, 3, 1] with a sampling time of 25 ms, best balancing time of 3.6 seconds, and an average balance time of 2 seconds. It used a set of three actions: [0N, -14N, +14N] and a forward prediction script.

4. An NFQ agent [6, 3, 1] with a sampling time of 50 ms, best balancing time of 3.6 seconds, and an average balance time of 2.2 seconds. It used a set of three actions: [-14N, 0N, +14N]

and avoided the use of forward prediction script. Instead, it used historical information to augment state information.

5. An NFQ agent [6, 6, 6, 1] with a sampling time of 50 ms, best balancing time of 2.67 seconds, and an average balance time of 2.12 seconds. It executed control using three actions: [-14N, 0N, +14N] and also avoided the use of forward prediction script similar to NFQ agent [6, 3, 1].

Initially, we intended to obtain better performing agents through GA. However, we later found out that an agent that was trained using GA from scratch was able to put a better performance than any other previously mentioned agents.

7.2 Experiments

Step 1: Optimize the best NFQ agents on the hardware using GA

Fundamentally, the GA performs crossover and mutation operations iteratively over a set of initial agents/networks called the 'population'. Therefore, we had to choose multiple agents with the same network structure, to be optimized using GA. We chose the NFQ agent [6, 3, 1] that had a sampling time of 50 ms using three actions: [-14N, 0N, +14N]. Additionally, we chose the top 2^{nd} and 3^{rd} best performing NFQ agents that were obtained while training NFQ agent [6, 3, 1]. We did not plan to use the offline DQN agent [4, 24, 16, 2] or any other configurations of NFQ agents as they made the use of a forward prediction script that made them a model-based controller.

We started with a population size of three NFQ agents with configuration [6, 3, 1] and a sampling time of 50 ms. These agents were allowed to play three episodes each on the hardware and the average score was calculated. This average score was considered as the fitness criteria of that agent. Next, we chose the top two agents based on their fitness levels and used them in the process of reproduction. These two agents were termed as the 'parents' for generating the next generation. In reproduction, we performed a crossover of the two parents followed by mutation to maintain diversity and produced five agents in total. These five agents were termed as the 'children'. With the inclusion of the parents, a total of seven agents formed the 2^{nd} generation. All the agents in the second generation were allowed to play three episodes each on the hardware and the average score of the generation was calculated. This average score was used to check the evolution of GA towards producing better agents. A sequence diagram describing the process used to optimize NFQ agents using GA on EV3 is as shown in figure 7.2.

We performed experiments comprising of 20 generations and the results are as shown in figure 7.1. Performing 20 generations produced a total of 73 different NFQ agents and the performance of the agents, in general, was not significant. A total of ≈5 hours of wall-clock training time was consumed. As seen from figure 7.1, the average balance time for each generation was fluctuating with a peak of 34 steps i.e. 1.7 seconds of average balance time in the 10^{th} generation. The possible reasons for the failure could be the small initial population size which started with 3 parents and comprised of just 7 children in each generation. It is generally observed that GA performs well with higher population size. As a result, we could not optimize the performance of the NFQ agent [6, 3, 1] using GA.

Step 2: Training agents from scratch using GA on simulation.

GA has been a popular evolutionary algorithm that generates high-quality solutions to optimization and search problems. We planned to implement the GA afresh with using any pretrained network unlike step 1. We started with a small architecture of shape [4, 4, 2] and implemented the GA algorithm on the Cartpole-v1 environment from the OpenAI Gym updated with hardware parameters. The initial population size was kept to 500. All the agents in a generation were played for 3 episodes on the cartpole-v1 environment and the average score attained by them symbolized their fitness level. Top 20 best GA agents were chosen as parents for mutation and crossover based

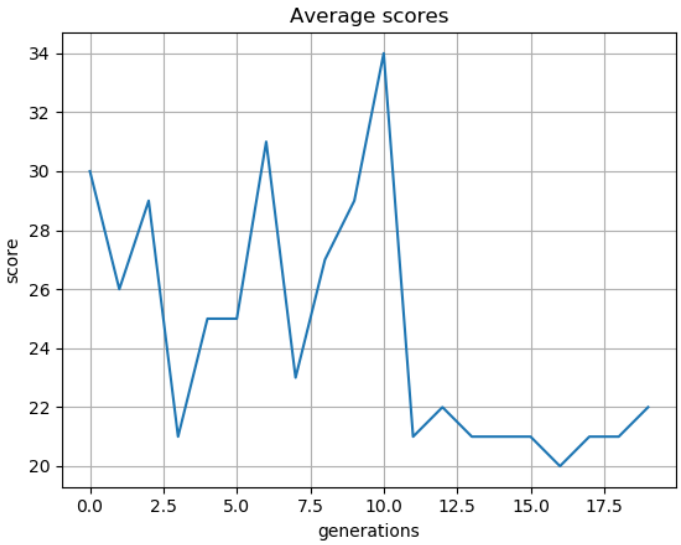


Figure 7.1: Performance of 20 generations in terms of average generation balance time.

upon the fitness scores. We produced 499 children agents as the outcome of the crossover and mutation process. To guarantee a progressive performance of the GA, we chose one elite to be carried over to the next generation from the current generation. The elite was chosen by retesting the average score of the top 5 parents out of the top 20 parents on the cartpole environment. The agent with the highest average score was then allowed to carry over to the next generation. The training was stopped at a point when the average scores of the top 50 agents (i.e. 10 % of the generation population) were 500 and the average scores of all the population were 90% of the highest achievable score (i.e. 500) in the cartpole-v1 environment. The training on simulation was successful and we chose a set of top 20 agents trained on simulation to train further on the hardware.

Step 3: Training trained GA agents from simulation on the hardware.

Obtaining trained GA agents from simulation had dual-motives. First, the GA could be used in its natural setting i.e. training on a dense population of agents, and second, training such a high number of agents on hardware was impractical due to time constraints. As a result, we had 20 trained GA agents which could still be managed to be trained on the hardware in the aspects of training time. The trained GA agents had the configuration [4, 4, 2] with the state $[x, \dot{x}, \theta, \dot{\theta}]$ at the input and two actions [-14N, +14N] at the output. The network configuration was significantly small with ReLU activation used in the hidden layers and sigmoid activation at the output. Further, the input was also not required to be normalized; hence, eliminating the z-score calculation and its corresponding computation time. As a consequence, we could observe a sampling time of merely 25 ms.

Table 7.1 depicts the performance of the trained GA agents on the hardware. The agent index column refer to the indexes of the agents when they were trained on the simulation. As observed, they indicate a comparatively better performance as compared to NFQ or DQN agents.

Moving further, we trained these agents for four generations on the hardware. The training process on the hardware followed a similar flow as shown in the sequence diagram 7.2, except that the initial population size was 20, 19 agents were produced through crossover and mutation using the top 10 parents, and one elite parent was added by retesting the top 10 best performing parents. We completed four generations of GA generating a total of 80 agents and training time

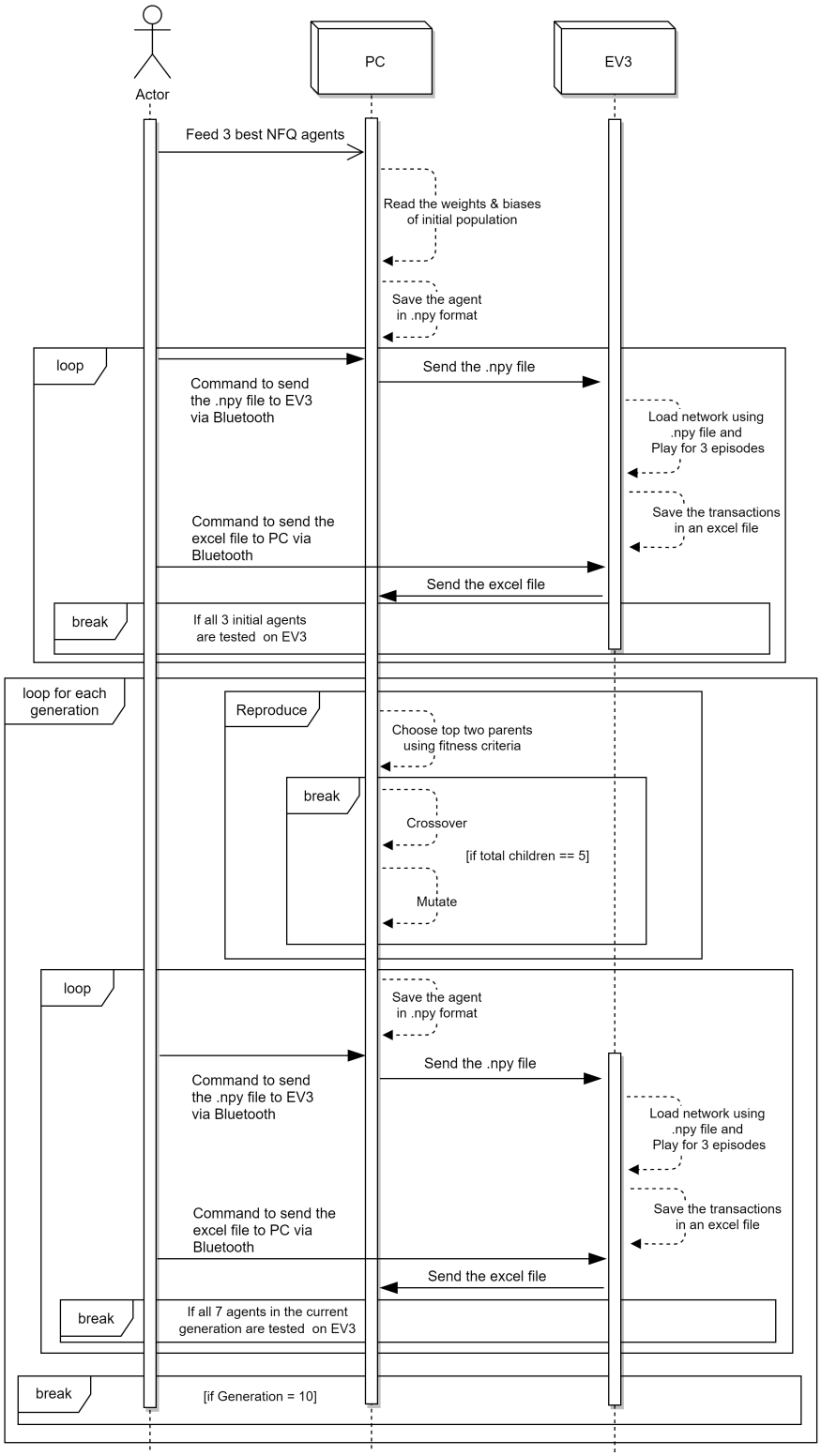


Figure 7.2: Sequence diagram describing the process of implementing GA to optimize NFQ trained agents on the hardware.

Agent Index	Balance Time (sec)	Agent Index	Balance Time (sec)
84	1.4	289	1.38
88	1.76	296	1.13
100	1.35	297	1.33
101	3.91	298	1.37
102	1.56	299	1.25
281	2.33	301	1.21
284	1.6	304	1.37
286	1.47	305	1.18
287	1.34	311	1.38
288	1.38	499	2.89

Table 7.1: Performance of the top 20 trained GA agents implemented directly from simulations on to the hardware.

of ≈20 hours. The results of training the trained GA agents from simulation on the hardware are as shown in table 7.2. We obtained an agent with index 45 that was able to balance for the highest amount of time of 5.7 seconds. Such a duration of balance time was never achieved by DQN or NFQ agents tested until this point. Agent 45 scored 98, 243, and 105 steps in three consecutive episodes on the hardware which accounted for an average score of 143 steps. Since this agent had the index 45, we term this GA agent as Agent-45 in the remaining section of the thesis. However, as seen from table 7.2, the average scores of the generation started to drop after the second generation which forced us to discontinue training further.

Generation 1		**Generation 2**		**Generation 3**		**Generation 4**	
Agent index	**Avg. Score**	**Agent index**	**Avg. Score**	**Agent index**	**Avg. Score**	**Agent index**	**Avg. Score**
0	51.66	30	54.33	60	41.66	90	50.33
1	59.33	31	56.33	61	35.33	91	46.33
2	51.70	32	57.33	62	50.66	92	46.66
3	51.70	33	46.33	63	44	93	61.33
4	70.33	34	84.66	64	45	94	48
5	52.33	35	53.33	65	45.33	95	47.33
6	112.66	36	105.66	66	44	96	62
7	55	37	47.66	67	39.33	97	47.33
8	56.33	38	56.66	68	45.66	98	45.33
9	52.66	39	92	69	43.66	99	51.66
10	50	40	50	70	41	100	46.66
11	43	41	46	71	47	101	51.66
12	48	42	50.66	72	39	102	59
13	59	43	52.66	73	49	103	54.33
14	50.33	44	42.33	74	44.66	104	47.66
15	52.66	45	143	75	45	105	47.33
16	52.66	46	47.66	76	45.33	106	53
17	46.33	47	103.33	77	40	107	52.66
18	51.33	48	59	78	44.33	108	47.33
19	63.33	49	80	79	51.66	109	54.33
Average	**53.9**	**Average**	**66.1**	**Average**	**43.8**	**Average**	**58.45**

Table 7.2: Results for training the trained GA agents [4, 4, 2] from the simulations on the hardware.

Step 4: Retraining the GA Agents with a modified version of GA and Gaussian noise

In the GA, it has been proved [23] that it can learn faster when the crossover function is excluded, keeping the mutation mechanism unchanged. We attempted to implement this version of GA in search of obtaining better agents than Agent-45. Also, it is important to note that the Agent-45 gave the best performance of balancing of 5.7 seconds only once and was not able to repeat it the next time. The possible issue could be that the pole's initialization could have also assisted the Agent-45 with a favorable starting point. In other words, we suspected the contribution of the state space drift in the performance of the Agent-45. Therefore, we decided to include Gaussian noise in the modified version of the GA implementation.

First, we trained agents from scratch on simulations using the modified version of GA. Then a set of 20 successfully trained agents was taken to hardware similar to the process done in step 2. These 20 agents were then trained for 8 generations on the hardware amounting to a total of 700 episodes (≈25 hours of training time). The training results for the GA agents on hardware are as shown in figure 7.3. We observed fluctuation in the average scores of each generation experimented on the hardware. Despite the addition of noise, this approach did not conclude with a better result than the Agent-45.

Agent	Score	Best	Scorers
Generation	0	Average	64.0
0	63.3	19	84.0
1	59.0	16	75.3
2	64.0	12	74.0
3	55.3	11	69.7
4	54.3	17	69.7
5	61.3	7	64.7
6	63.0	2	64.0
7	64.7	18	63.7
8	63.0	0	63.3
9	60.0	8	63.0
10	56.0	6	63.0
11	69.7	5	61.3
12	74.0	14	61.0
13	59.0	15	60.0
14	61.0	9	60.0
15	60.0	13	59.0
16	75.3	1	59.0
17	69.7	10	56.0
18	63.7	3	55.3
19	84.0	4	54.3
Generation	1	Average	56.9
0	52.7	9	77.7
1	51.3	14	68.7
2	60.0	5	63.3
3	57.0	15	63.0
4	51.7	8	61.0
5	63.3	10	61.0
6	51.7	17	60.0
7	54.7	2	60.0
8	61.0	3	57.0
9	77.7	7	54.7
10	61.0	11	54.3
11	54.3	19	52.7
12	52.7	12	52.7
13	52.7	13	52.7
14	68.7	0	52.7
15	63.0	6	51.7
16	48.0	4	51.7
17	60.0	1	51.3
18	43.7	16	48.0
19	52.7	18	43.7

Generation	2	Average	55.5
0	56.0	15	67.0
1	49.3	16	65.7
2	52.3	12	64.7
3	62.0	6	64.0
4	46.7	3	62.0
5	52.7	7	58.3
6	64.0	14	57.3
7	58.3	0	56.0
8	50.0	13	55.3
9	51.7	17	55.0
10	47.3	18	53.0
11	49.3	5	52.7
12	64.7	19	52.7
13	55.3	2	52.3
14	57.3	9	51.7
15	67.0	8	50.0
16	65.7	11	49.3
17	55.0	1	49.3
18	53.0	10	47.3
19	52.7	4	46.7
Generation	3	Average	63.4
0	56.7	7	85.3
1	69.0	18	83.3
2	72.0	5	81.3
3	67.0	2	72.0
4	52.7	11	70.3
5	81.3	1	69.0
6	57.3	3	67.0
7	85.3	13	62.7
8	55.7	16	60.3
9	53.0	19	59.7
10	59.3	10	59.3
11	70.3	17	58.3
12	56.3	6	57.3
13	62.7	0	56.7
14	54.0	12	56.3
15	54.3	8	55.7
16	60.3	15	54.3
17	58.3	14	54.0
18	83.3	9	53.0
19	59.7	4	52.7

Generation	4	Average	51.6
0	48.7	9	69.0
1	47.0	2	61.0
2	61.0	6	60.3
3	45.7	5	58.7
4	49.7	18	56.3
5	58.7	17	55.7
6	60.3	8	53.3
7	48.3	16	51.7
8	53.3	10	50.7
9	69.0	13	50.3
10	50.7	4	49.7
11	49.0	11	49.0
12	43.3	0	48.7
13	50.3	7	48.3
14	43.0	19	47.3
15	43.7	1	47.0
16	51.7	3	45.7
17	55.7	15	43.7
18	56.3	12	43.3
19	47.3	14	43.0
Generation	5	Average	55.0
0	51.7	10	75.3
1	56.0	4	71.3
2	61.0	14	65.7
3	44.0	7	63.7
4	71.3	2	61.0
5	55.3	12	58.0
6	47.0	9	57.0
7	63.7	1	56.0
8	49.3	11	55.7
9	57.0	5	55.3
10	75.3	0	51.7
11	55.7	16	50.0
12	58.0	8	49.3
13	48.3	19	49.3
14	65.7	18	48.7
15	46.7	13	48.3
16	50.0	6	47.0
17	45.7	15	46.7
18	48.7	17	45.7
19	49.3	3	44.0

Generation	6	Average	48.9
0	46.0	8	80.0
1	79.7	1	79.7
2	68.0	10	74.7
3	38.3	2	68.0
4	68.0	4	68.0
5	38.3	14	54.7
6	38.3	19	50.3
7	37.3	0	46.0
8	80.0	15	40.3
9	39.3	9	39.3
10	74.7	12	39.0
11	38.3	16	39.0
12	39.0	11	38.3
13	36.7	6	38.3
14	54.7	5	38.3
15	40.3	3	38.3
16	39.0	7	37.3
17	35.3	18	37.0
18	37.0	13	36.7
19	50.3	17	35.3
Generation	7	Average	51.5
0	47.3	6	70.7
1	57.3	16	68.3
2	41.0	1	57.3
3	55.0	3	55.0
4	46.3	9	53.7
5	47.7	7	53.7
6	70.7	17	51.0
7	53.7	10	51.0
8	51.0	8	51.0
9	53.7	14	50.7
10	51.0	12	50.7
11	47.7	15	50.0
12	50.7	5	47.7
13	46.3	11	47.7
14	50.7	0	47.3
15	50.0	19	46.7
16	68.3	4	46.3
17	51.0	13	46.3
18	44.0	18	44.0
19	46.7	2	41.0

Generation	8	Average	49.6
0	51.0	19	79.7
1	45.7	12	63.0
2	48.3	18	63.0
3	45.3	15	55.3
4	42.7	11	52.0
5	41.3	0	51.0
6	44.0	9	49.3
7	44.7	10	49.0
8	43.7	2	48.3
9	49.3	13	45.7
10	49.0	16	45.7
11	52.0	1	45.7
12	63.0	3	45.3
13	45.7	7	44.7
14	41.0	6	44.0
15	55.3	8	43.7
16	45.7	17	42.7
17	42.7	4	42.7
18	63.0	5	41.3
19	79.7	14	41.0

Figure 7.3: Hardware results for training agents obtained from the modified GA through simulations.

7.3 Results

Following are the key results obtained by implementing the Genetic Algorithm in controlling a real-life cartpole system:

- An NFQ agent [6, 3, 1] was attempted to be optimized by using GA on top of the trained agent. However, GA was unable to optimize the performance of the NFQ agent [6, 3, 1].
- The best AI controller among all the approaches used in this thesis to control the real-cartpole system was obtained when a GA algorithm was used to train agents first through simulation and then followed by additional training on the hardware. The Agent-45 was able to balance the real cartpole system for a maximum of 5.7 seconds and an average of 2.42 seconds.

7.4 Conclusion

The use of the Genetic Algorithm for balancing the cartpole was successful. One of the key reasons for its success, despite failures of DQN and NFQ algorithms, was the algorithm itself. The point in time at which the GA agent produced the action did not require the condition that the input state should also be at the same point in time. On the other hand, DQN and NFQ heavily relied on the fact that there should be almost no delay between the input state and the action produced by the agent. Nevertheless, NFQ had incorporated the historical information (i.e. previous action a') as a countermeasure for the delays, yet did not produce a controller with a performance of a satisfactory level.

Another key observation was that, despite the use of elitism to guarantee a better quality of agents with increasing generations, the GA algorithm was not producing higher quality agents (referring to the average scores for the generations in the table 7.2 and figure 7.3). This can be attributed to the naive design of the cartpole system. The state space drift and non-real-time response nature of the hardware had a major contribution in losing some of the best agents obtained in the previous generation to the next generation. To exemplify, Agent-45 was obtained in the 1^{st} generation with the index 45 in which its average score was 143 (Per episode score: 98, 243, 105). However, the agent could not give the same performance in the tests for choosing the elite candidate to carry over to the next generation. As a result, it got left out despite showing the best performances in the initial trials. Nevertheless, after the completion of training, we tested the performance of the Agent-45 and observed that it maintained an average balance time of 2.42 seconds and also could reach up to a balanced time of a maximum of 6.03 seconds.

Therefore, we chose the Agent-45 as the solution of the 'DNN controllers' category and would optimize it through the Pruning technique described later in chapter 9. In the next chapter, we design and implement an 'SNN controller' as the second category of AI controllers.

Chapter 8

Implementation: SNN

With Agent-45 representing the category of 'DNN controllers', we describe how we used the Genetic Algorithm in building SNN category controller in this chapter. The outcome of this chapter is the SNN controller Agent-87 that executes an average balance time of 2.48 seconds on the real-cartpole system.

8.1 Motivation

Genetic algorithm is originally a searching algorithm that finds an optimal solution to a given problem. Since it showed a better performance for developing DNNs for our cartpole problem, we chose to use the same algorithm to develop the SNN controller. In particular, we used the genetic algorithm excluding crossover and including mutation to train the SNN agents.

8.2 Experiments

Step 1: Training the SNN agents from scratch using GA on the simulation.

We adopted a similar technique that we used in producing the DNN agents by initially training the SNN agents using GA from scratch on the simulation. The simulation training setup was exactly similar to that of training DNNs through GA. The architecture of the SNN agent was chosen to be [4, 4, 2] with a ReLU activation used in the hidden layer whereas sigmoid activation in the output layer. The target platform on the simulations was the Cartpole-v1 environment from the OpenAI Gym befitting the hardware parameters. Training of the SNN agents started with a population size of 500. The agents were allowed to play 3 episodes and the average score attained by them was considered as their fitness score. In each generation, the top 20 agents were considered for reproducing the next generation of 499 agents through mutation. An elite agent from the top 5 parents was carried over to the next generation forming a total population of 500 in the new generation. We considered the training as completed when the average scores of the top 50 agents were 500 and that of the entire generation was 90% of the maximum achievable score i.e. 90% of 500 = 450. On completion of the training on simulation, the top 20 agents were carried over to the hardware for further training.

Step 2: Training the trained SNN agents using GA on the hardware.

SNN agents primarily differentiate from the DNN agents in terms of the neuron structure. SNN neurons, termed as LIF neurons, contain a form of memory that stores each incoming pulse. Each LIF neuron has a fixed memory potential threshold that decides the firing point of that neuron. With each incoming spike, the potential of the neuron increases until it reaches the threshold, after

which the neuron fires a spike and clears its internal memory through discharging its membrane potential. This entire process comes at the cost of additional calculations that the SNN agent does for keeping a track of the membrane potential of each of its neurons. Therefore, owing to this overhead of calculations, the sampling time of the SNN agent also increases and we observed it to be 35 ms on an average on the hardware. With the sampling time set to 35 ms, we performed experiments producing 5 generations of SNN agents starting with an initial population of 20 agents on the hardware. The training process that we followed was similar to the sequence diagram as shown in figure 7.2.

In every generation, 19 agents were reproduced using the top 2 parents, and one elite agent was added to the generation that was chosen by retesting the top 2 best performing parents. At the end of training on the hardware, we produced a total of 110 agents requiring a training time of ≈25 hours. The results of the training of the SNN agents using GA on hardware is as shown in figure 8.1.

Generation 1		**Generation 2**		**Generation 3**		**Generation 4**		**Generation 5**	
Agent index	**Avg. Score**	**Agent index**	**Avg. Score**	**Agent index**	**Avg. Score**	**Agent index**	**Avg. Score**	**Agent index**	**Avg. Score**
0	44.3	22	77.7	45	59	68	44.3	91	52.7
1	65.3	23	47.3	46	66	69	51.7	92	57.3
2	77	24	50	47	53	70	53.7	93	53.7
3	50	25	51	48	63.7	71	60	94	116.3
4	47.7	26	82.3	49	61	72	50.7	95	52
5	47.7	27	72.7	50	49.7	73	120.3	96	55.3
6	52.3	28	52	51	79	74	82.7	97	59.7
7	78	29	50.3	52	53	75	48	98	56.7
8	111.7	30	110.7	53	76.7	76	100.3	99	60.3
9	52.7	31	55.3	54	63.3	77	47	100	64
10	65	32	76.7	55	52.3	78	73.7	101	85.3
11	85.7	33	51	56	54.3	79	56	102	56.3
12	46.7	34	85	57	61.3	80	48.7	103	112.7
13	46.7	35	59.7	58	45.7	81	48	104	54
14	81.7	36	56.7	59	52.3	82	51	105	68.3
15	103	37	47	60	80.3	83	72.3	106	76.7
16	65	38	56	61	45.3	84	50	107	45.7
17	64.3	39	81.7	62	50	85	55.7	108	67.7
18	106.7	40	69.7	63	73.3	86	61.7	109	50.3
19	61.3	41	70.7	64	106	87	122.3	110	52.3
Average	**67.64**	**Average**	**65.18**	**Average**	**62.26**	**Average**	**64.905**	**Average**	**64.86**

Table 8.1: Results for training the trained SNN agents [4, 4, 2] using GA from the simulations on the hardware.

We could obtain an agent in the 4^{th} generation with index 87 that could get the highest average score of 122.3. Furthermore, the average scores per generation were also decreasing gradually. Therefore we selected the Agent-87 as the representative of the SNN category of controllers in this thesis. The average score attained by the agent when we tested it later for 30 episodes was 2.48 seconds with the best score of 3.6 seconds. Thus, the maximum balance time achieved by the SNN category of controllers was 2.48 seconds on the cartpole system.

8.3 Results

We obtained an SNN agent that was trained using GA from scratch initially on the simulations followed by training on the hardware. The agents that were trained on the simulations were able to balance the pole completely on the Cartpole-v1 environment with the hardware parameters.

Additional training of the trained agents from the simulation on the hardware produced an Agent-87 as the best performing SNN controller. It could balance the pole for an average of 2.48 seconds of balance time and with a maximum of 3.6 seconds. The configuration of the SNN agent is exactly similar to the DNN Agent-45 which is [4, 4, 2].

8.4 Conclusion

The Genetic Algorithm once again proved to be a better search method to build SNN controllers in controlling a real-time practical cartpole system. However, the end performance of the SNN controller could not exceed above 2.5 seconds in terms of average balance time. As mentioned earlier, the key reason for the failure of the agents is the non-real-time response of the hardware. The fact that the Linux based OS provides a non-real-time response, could be a major reason for the failure of the algorithms like DQN and NFQ as well. We discuss the inconsistencies provided by the EV3 hardware in chapter 10 in detail.

In conclusion, we obtained an SNN controller named Agent-87 using GA that could be used to balance the real-life cartpole system for an average balance time of 2.48 seconds.

Chapter 9

Optimization

In this chapter, we understand how optimization plays a vital role in the performance of the AI controller. We describe how the optimization of a neural network is directly related to its energy efficiency. Furthermore, we discuss the effect of pruning neurons on the best performing DNN and SNN agents. We highlight the simulation results of pruning followed by its hardware implementation for both the AI controllers. The key insights we provide through this chapter is that the performance of controllers deteriorates as we reduce the number of neurons in its architecture.

Optimization of network architecture is one of the key aspects as it directly affects the performance of the network. The thesis aims at designing AI controllers such that they are energy efficient for a particular control application. The energy consumption of the controller is directly proportional to the architecture of the neural network. If we have a larger model (i.e. more layers and/or neurons), it would require more number of parameters i.e. weights and biases to calculate the action. In our case both the agents: Agent-45 from the DNN category and Agent-87 from the SNN category have the configuration [4, 4, 2] i.e 10 neurons. The parameters of each of the agents are as shown in table 9.1.

	Network Parameters							
	DNN Agent-45				**SNN Agent-87**			
W[0]	0.114611	-0.36976	0.004006	-0.01454	0.497691	-0.39565	-1.02451	-0.95091
	-0.1884	0.313377	-0.60841	-0.53499	0.14929	-0.58698	0.09589	-0.51850
	0.296651	0.190262	-0.66778	-1.17585	0.82563	-2.33728	-0.08185	-2.41357
	-0.75604	-0.69925	-0.69153	-2.42713	0.92475	-1.02903	0.63633	-1.52933
B[0]	-0.38316	0.087548	0.206646	0.816	0.45309	0.71787	0.56929	0.46604
W[1]	0.153732	0.507467			-1.39897	1.10257		
	0.617005	0.121731			1.24334	-2.90878		
	0.223229	-0.85645			-1.50185	1.712698		
	1.451527	-1.86267			1.21840	-0.58915		
B[1]	-0.9865	1.917096			-0.21924	0.155174		

Table 9.1: DNN and SNN agents' network parameters

Thus, there are a total of 30 network parameters used by each of the agents to produce one action. During the execution, the weights and biases of the agent are initially stored in the memory of the EV3 brick. Whenever a state is fed at the input of the neural network, the weights and biases are accessed from the memory to the processor for calculations such as additions and multiplications. These weights and biases are stored in the memory of EV3. For a generic platform

with a 45 nm CMOS process, we consider the energy cost for basic arithmetic calculations and memory accesses as shown in figure 9.1. Memory accesses from the DRAM are energy expensive by three orders of magnitude as compared to that of simple arithmetic operations. As a result, the network that has a higher number of parameters that are required to be stored in memory away from the processor requires a higher number of memory references to perform the calculations, which in turn results in higher energy consumption. For a real-time task like balancing a pole on a resource-constrained device like EV3 brick, a model with a larger set of parameters and more number of computations becomes a bottleneck. Therefore, we aim to optimize the network by reducing its parameters such that its performance remains unaltered or deteriorates minimally.

Operation	Energy [pJ]	Relative Cost
32 bit int ADD	0.1	1
32 bit float ADD	0.9	9
32 bit Register File	1	10
32 bit int MULT	3.1	31
32 bit float MULT	3.7	37
32 bit SRAM Cache	5	50
32 bit DRAM Memory	**640**	**6400**

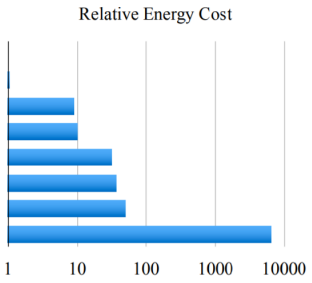


Figure 9.1: Energy table for 45 nm CMOS process [12]

9.1 Pruning

One of the common approaches in optimizing the network architecture is Pruning. In that, we trim the network parameters that are redundant and can be eliminated while maintaining network performance. There exist several pruning algorithms [6] to choose the set of weights and biases, and eventually, neurons that can be removed leaving the performance unharmed. Most of the pruning techniques are suitable for networks of a very large size i.e. > 100 neurons. Since the network architecture that we are using is already small i.e. 10 neurons, we choose to do the pruning process manually as opposed to implementing complex pruning algorithms. Traditionally, the process of pruning is a three stages process as shown in figure 9.2. In our case, we used GA as the algorithm to produce the DNN and SNN agents. Retraining the GA agents is difficult due to the mutation process involved during the production of the next generation. Thus, we limit the pruning process to simply eliminating the unnecessary neurons and observe the performance of the agent.

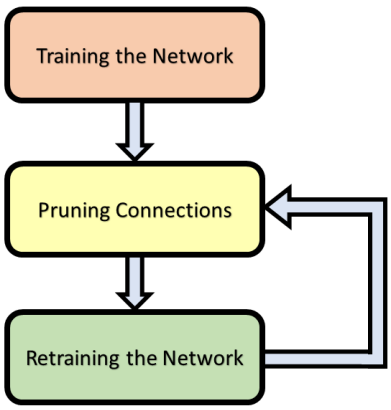
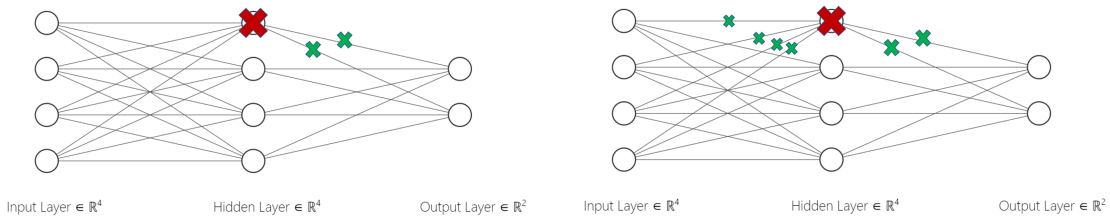


Figure 9.2: Traditional 3-step pruning process

We performed the pruning of our agents in a naive fashion in which we dropped neurons one-by-one and tested the performance of the agent. Due to the absence of retraining, the pruning technique that we used could also result in a deterioration of the performance of the network. Thus, if the performance after pruning one or more neurons has not dropped substantially or if it has improved, we continued dropping neurons further and repeated the process. Initially, we started dropping neurons one at a time from the hidden layer by setting the weights (also called synapses) to that corresponding neuron to zero. As seen from figure 9.3a the weights with a green cross are set to zero thereby imitating that the neuron with the red cross is removed/absent. By removing one neuron from the hidden layer, we can also remove the weights connected to that neuron from the previous layer that becomes redundant as shown in figure 9.3b.



(a) Technique for pruning synapses and neurons

(b) Network structure after pruning of 1 neuron

Figure 9.3: Naive pruning methodology used to prune DNN and SNN agents.

It was necessary to validate if the discussed approach would help in successfully reducing the structure of the network. Therefore, we performed the pruning of each of the agents on simulation, noted the results, and then performed the pruning process on the hardware.

Terminology:

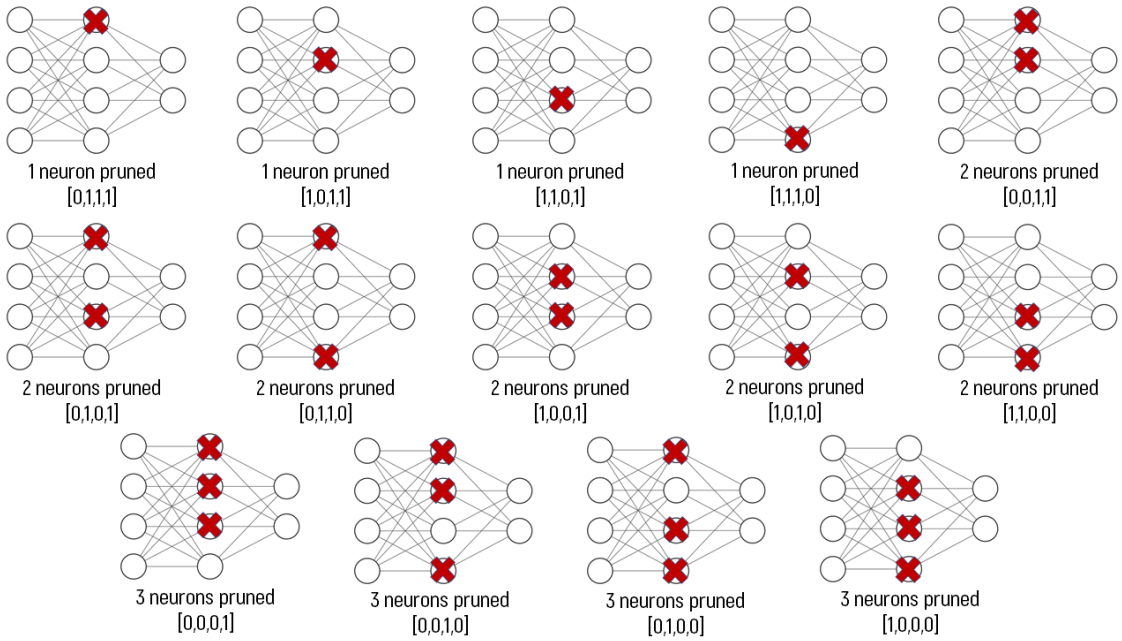


Figure 9.4: Notation for indicating which neuron is pruned during the experiments.

The scope of pruning in our network is focused on the hidden layer that contains four neurons. There exist 16 different possible combinations of pruning neurons in the hidden layer with four

neurons. Note that there has to be at least one neuron in the hidden layer to learn the nonlinearity in the cartpole problem. We denote the pruning configurations as [x_1, x_2,x_3, x_4] where x_n denotes the n^{th} neuron from the top to bottom in the hidden layer of the network. If the n^{th} neuron from the hidden layer is pruned, the corresponding x_n is substituted by '0'. Thus the network without pruning would be denoted as [1, 1, 1, 1]. The possible combinations of pruning and their notations are as shown in figure 9.4.

9.2 Pruning DNN controller

Agent-45 was obtained using the Genetic algorithm and was able to execute control over the cartpole system for 2.42 seconds of average balance time. This controller was obtained by first training it on the simulations followed by training on the hardware. To begin pruning the Agent-45, we first had to check if it was still able to balance the pole on the simulated cartpole system. The simulation setup was the cartpole-v1 environment replaced with the hardware parameters. The results of Agent-45 performing on simulations is as shown in figure 9.5. The DNN agent was able to balance the pole on the simulations completely for a test of 100 episodes each lasting for 500 steps.

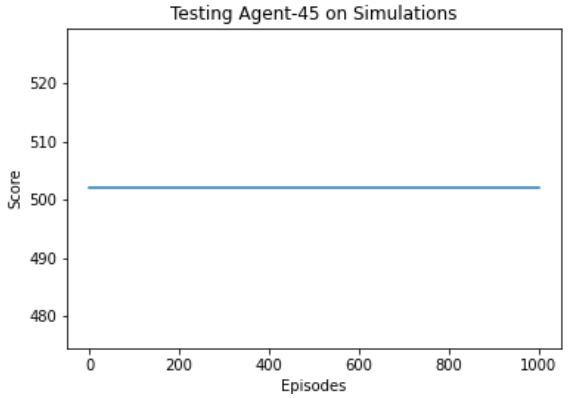


Figure 9.5: Performance of the Agent-45 on simulation over 1000 episodes.

9.2.1 Simulation results for pruning DNN

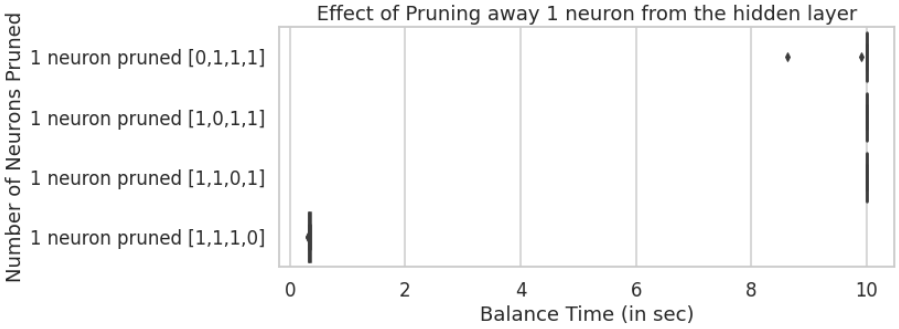


Figure 9.6: Performance of the Agent-45 after pruning 1 neuron from the hidden layer on simulations.

The simulation setup involved testing the agent with different configurations as shown in figure 9.4. The test involved 10 episodes with a maximum allowed balance time of 10 seconds and the

average balance time was calculated. We began with pruning a single neuron from the hidden layer. There were four different ways in which a single neuron could be pruned and they are as shown by the y-axis in the figure 9.6. We observed that the Agent-45 was able to balance the pole for the maximum allowable time for all configurations except when the last neuron from the hidden layer was pruned. The plot in figure 9.6 is a box-and-whisker plot that has got squeezed into a vertical line implying that the mean, maximum, minimum, and the quartiles are all the same. The additional points in the case of [0, 1, 1, 1] and [1, 1, 1, 0] configurations describe the outliers based on a method used by Seaborn library that is a function of the inter-quartile range [1].

Next, we pruned two neurons from the hidden layer of Agent-45 in six different ways as shown in figure 9.7. In this case, we observed a similar pattern of performance as found in pruning a single neuron. The agent was unable to balance the pole when the last neuron in the hidden layer was pruned. This is indicated by the nearly zero average balance time for configurations [0, 1, 1, 0], [1, 0, 1, 0] and [1, 1, 0, 0].

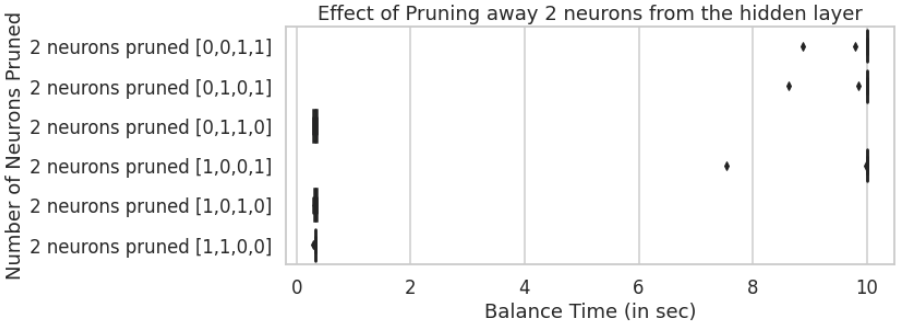


Figure 9.7: Performance of the Agent-45 after pruning 2 neurons from the hidden layer on simulations.

Lastly, we experimented with pruning three neurons from the hidden layer in four different ways and obtained the results as shown in figure 9.8. Combining the results for pruning three neurons with the previous pruning results, we observed that the last neuron in the hidden layer was enough to be able to balance the pole on the simulations for the maximum balance time. In other words, a DNN controller with a configuration of [4, 1, 2] i.e. 7 neurons is fully capable of controlling the cartpole system.

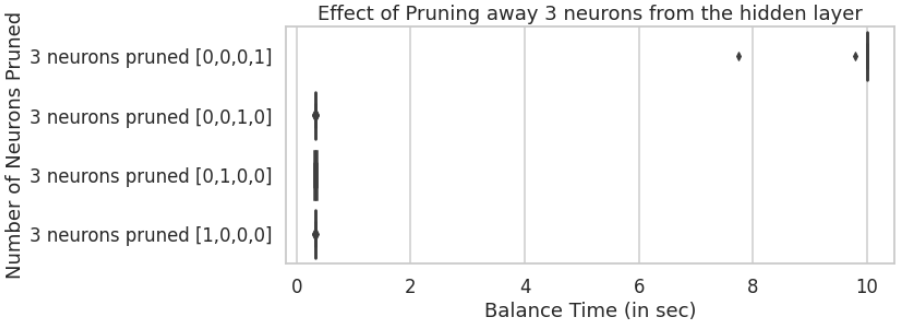


Figure 9.8: Performance of the Agent-45 after pruning 3 neurons from the hidden layer on simulations.

9.2.2 Summary of pruning DNN controller on the simulations

Pruning the Agent-45 on the simulations was fruitful as it verified the advantage of pruning in reducing the network structure and also provided key insights on the important connections/neurons in controlling the cartpole system. The Agent-45 was able to perform equally well despite the removal of first three neurons in its hidden layer because the weights and biases associated to that neuron were comparatively higher than the other neurons (see table 9.1: 4^{th} column under 'DNN Agent-45'). We plot the best configurations of Agent-45 when one, two, and three neurons are pruned, in the figure 9.9 and the performance remains unaltered.

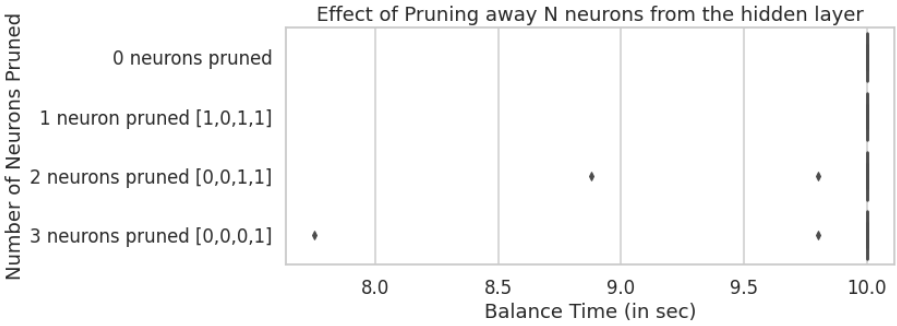


Figure 9.9: Performance of the Agent-45 after pruning n neurons from the hidden layer on simulations.

9.2.3 Hardware results for pruning DNN

In this section, we perform the pruning operation similar to the simulations, on the hardware. The only difference was the number of the test episodes carried out per configuration. We performed 10 experiments for each of the possible pruning configurations owing to the time constraints of testing episodes on the hardware. The results are the box-and-whisker plots of the balance time for each pruned configuration of the Agent-45.

The effect of choosing a single neuron in four different ways to be pruned from the hidden layer on the hardware is shown in figure 9.10. We observed that dropping the first neuron from the hidden layer gave a better performance as compared to dropping any other neuron in that layer. Further, this result was also in compliance with the simulation result of pruning single neuron such that the absence of the last neuron in the hidden layer results in a failure of the DNN controller.

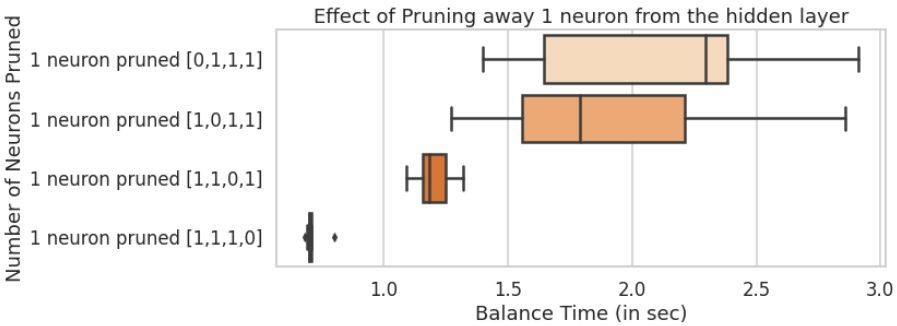


Figure 9.10: Performance of the Agent-45 after pruning 1 neuron from the hidden layer on the hardware.

As discussed earlier, choosing two neurons to be pruned from the hidden layer can be done in

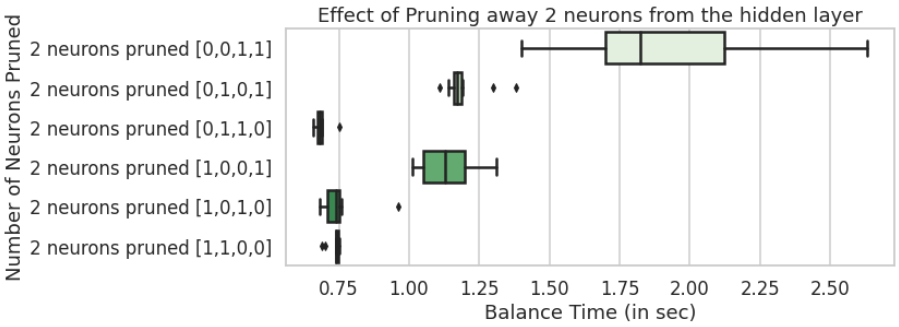


Figure 9.11: Performance of the Agent-45 after pruning 2 neurons from the hidden layer on the hardware.

six different ways. The results for pruning two neurons with all possible combinations are as seen in figure 9.11. It was observed that dropping the first and second neurons from the hidden layer gave a better performance as compared to dropping any other pair of neurons. It is also important to focus on the configurations that pruned the last neuron where the controller had failed completely.

Lastly, we plot the performance of the Agent-45 when 3 neurons were pruned from its hidden layer in four different combinations. Experiments showed that the Agent-45 could still perform well using only the last neuron from the hidden layer (see figure 9.12).

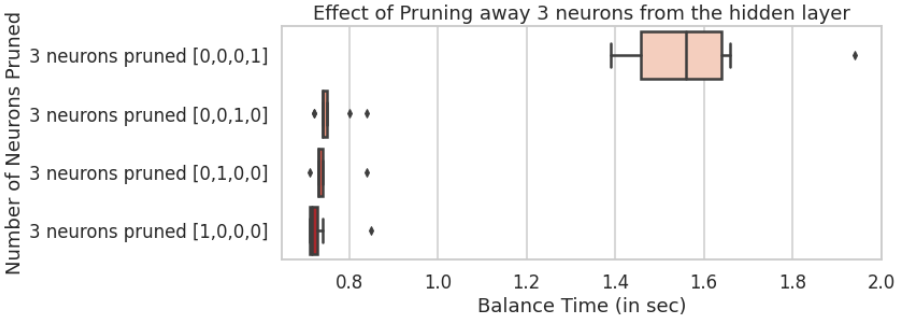


Figure 9.12: Performance of Agent-45 after pruning 3 neurons from the hidden layer.

9.2.4 Summary of pruning DNN controller on the hardware

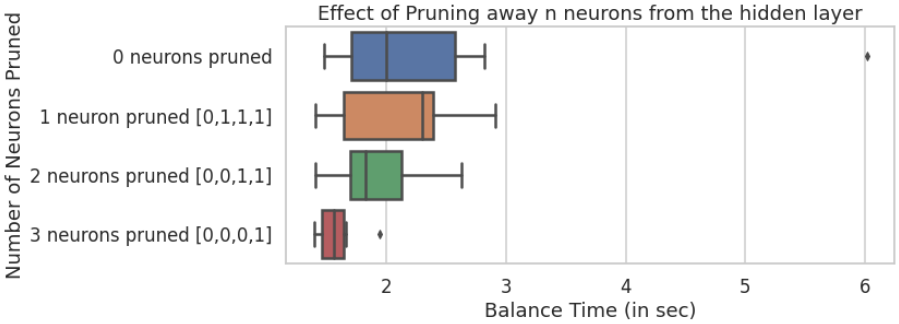


Figure 9.13: Summary of pruning neurons in the Agent-45 on the hardware.

We choose the best configuration of pruning one, two, and three neurons from figures 9.10,

9.11 and 9.12 respectively and compared it with the performance of Agent-45 where no neurons are pruned. This comparison is seen in figure 9.13 where we observed that the performance of Agent-45 decreases gradually as we prune neurons from the hidden layer. Overall, the performance graph of the Agent-45 follows the simulations in terms of the importance of the last neuron in the hidden layer due to its corresponding weights and biases. The last column of the DNN Agent-45 network parameter in table 9.1 indicates the weights and biases that are connected to the fourth neuron in the hidden layer. Further, the last row i.e. W[1] matrix signifies the weights that connect the fourth hidden neuron to the output layer. These values are comparatively larger than other connections which make the DNN controller highly reliable on the last neuron in its hidden layer.

9.3 Pruning SNN controller

Similar to the DNN controller, the SNN controller: Agent-87 was obtained using the Genetic algorithm and was able to execute control over the cartpole system for 2.48 seconds of average balance time. The controller was also obtained by initially training it on the simulations and then training it on the hardware. Before pruning the Agent-87, we implemented the agent unpruned, on the simulations, to check if it was still able to balance the pole. The simulation setup was similar to that used for the DNN controller as discussed in section 9.2. It was conclusive from the results shown in figure 9.14 that the SNN controller was able to balance the pole completely on the simulations.

Figure 9.14: Performance of the Agent-87 on simulation over 1000 episodes.

9.3.1 Simulation results for pruning SNN

The simulation setup that we used for testing was exactly similar to that of DNN. We performed the same set of pruning configurations for the SNN controller and recorded the results. The simulation test consisted of 10 episodes that allowed a maximum balance time of 10 seconds. We started the pruning of the Agent-87 by removing single neuron in four different ways. The result of the experiment is shown in figure 9.15. We observed that, in the presence of both the neurons: first and second from the hidden layer, the SNN controller performed the best.

Next, we proceeded with pruning two neurons from the hidden layer in six different methods and the results are shown in figure 9.16. In this case, we could observe the pattern followed by the SNN controller as seen earlier. The Agent-87 was able to balance the pole for the maximum allowable time when its configuration involved the first and second neurons in the hidden layer.

Lastly, a three neuron pruning was done on the Agent-87 on the simulation and the results obtained are shown in figure 9.17. In this case, since no two neurons could be left unpruned in the hidden layer, the agent could perform better with the pruning configuration of [0, 0, 0, 1] as compared to the remaining configurations. However, the performance of this configuration was worse than the results obtained from pruning one and two neurons.

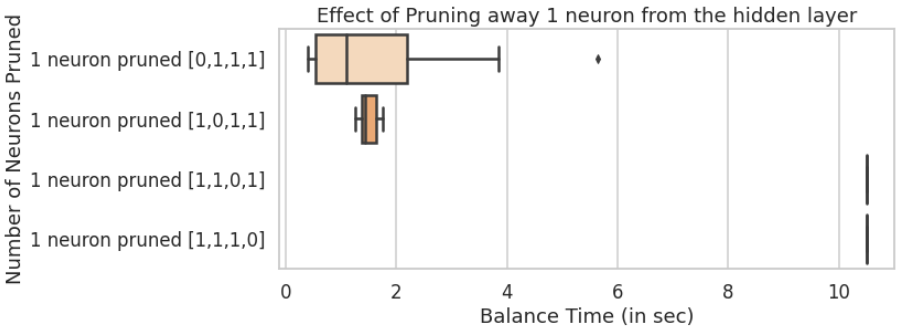


Figure 9.15: Performance of Agent-87 after pruning 1 neuron from the hidden layer on simulations.

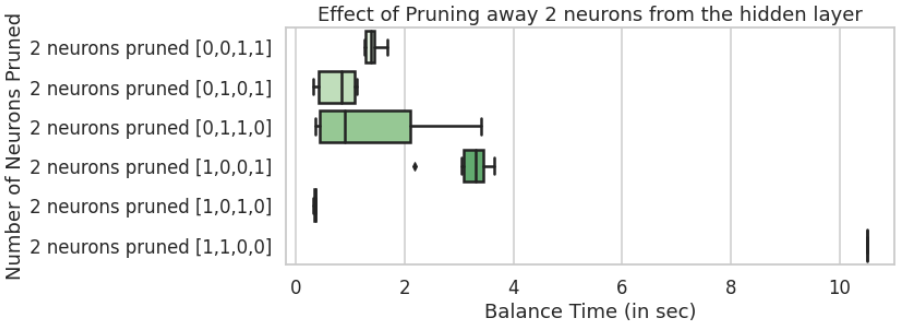


Figure 9.16: Performance of Agent-87 after pruning 2 neurons from the hidden layer on simulations.

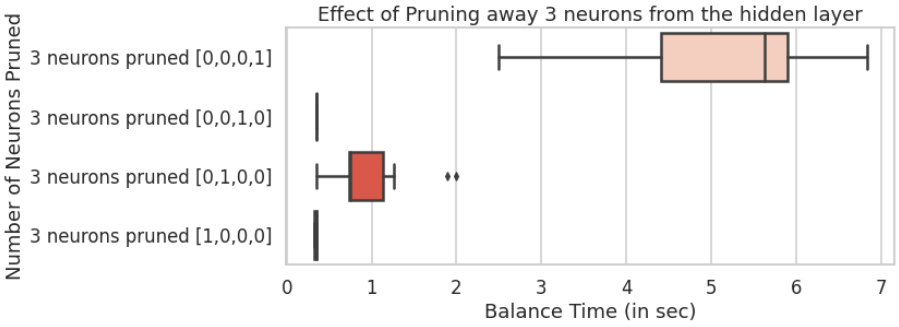


Figure 9.17: Performance of Agent-87 after pruning 3 neurons from the hidden layer on simulations.

9.3.2 Summary of pruning SNN controller on the simulations

SNN Agent-87 has a comparatively uniform distribution of weights as compared to DNN Agent-45. This is evident from the SNN Agent-87 column of the network parameter in table 9.1. Almost all the weights and biases of the network have their values distributed evenly with a range. Thus, the important neurons/parameters cannot be directly observed using the network parameter table. However, SNN executes a better performance when either the first two neurons or only the last neuron from the hidden layer is present. This can be concluded from the overall results obtained from the pruning of the SNN agent on the simulation (see figure 9.18).

9.3.3 Hardware results for pruning SNN

We repeated the pruning procedure on the hardware with the change similar to that of pruning DNN on the hardware. Box-and-whisker plots describing the performance of the pruned Agent-87

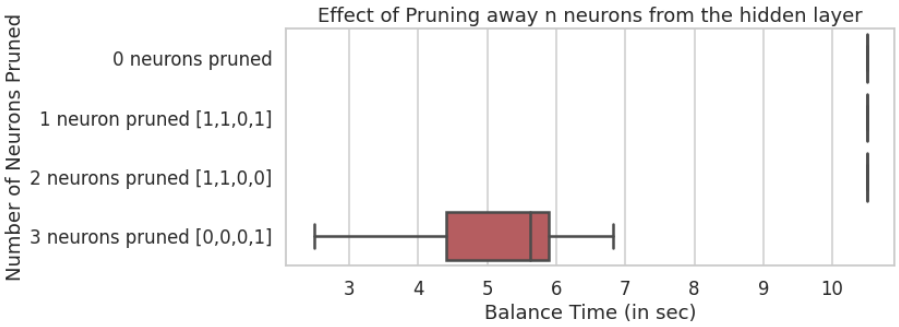


Figure 9.18: Performance of Agent-87 after pruning n neurons from the hidden layer on simulations.

are plotted in this section and the summary of pruning is provided at the end.

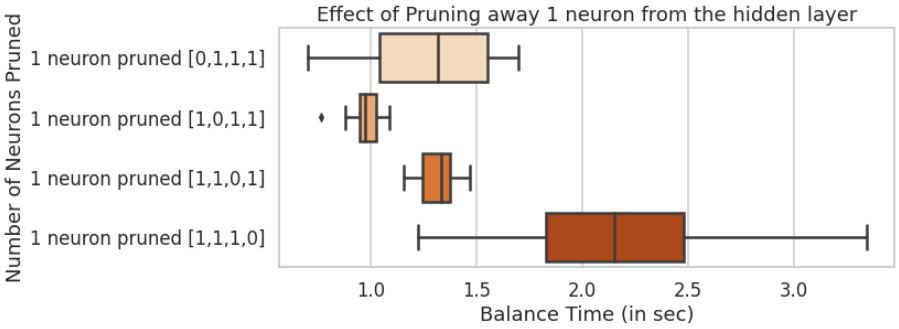


Figure 9.19: Performance of Agent-87 after pruning 1 neuron from the hidden layer on the hardware.

We started with pruning a single neuron in the hidden layer of Agent-45 and observed that it gave its best performance when only the last neuron was pruned [1, 1, 1, 0] (see figure 9.19). This was in alignment with the simulation result of pruning a single neuron.

Next, we pruned two neurons from the hidden layer of the Agent-87 in six different combinations and the result is shown in figure 9.20. Similar to the simulations, the SNN controller displayed better versions of control with the configurations [0, 1, 1, 0] and [1, 1, 0, 0].

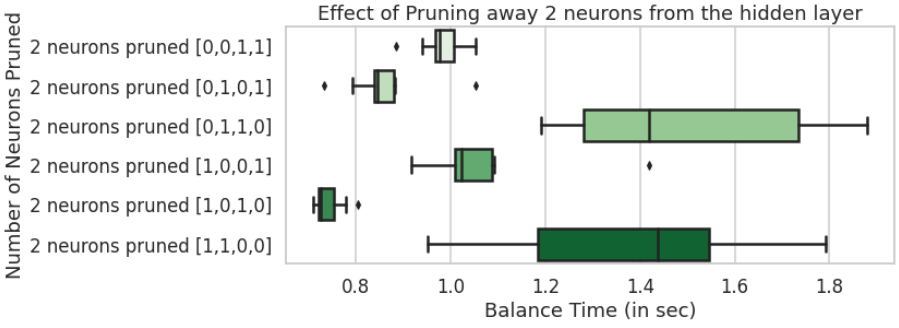


Figure 9.20: Performance of Agent-87 after pruning 2 neurons from the hidden layer on the hardware.

Finally, a three neuron pruning was performed in four different ways on the SNN controller. The hardware performance of the Agent-87 as shown in figure 9.21 which proved an exactly similar

behavior as that of simulations.

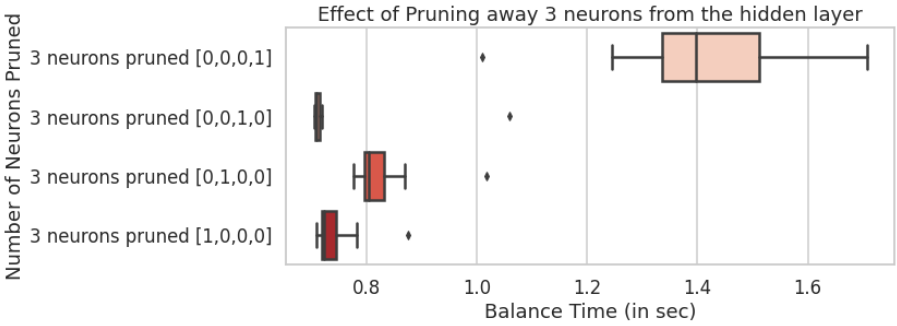


Figure 9.21: Performance of Agent-87 after pruning 3 neurons from the hidden layer on the hardware.

9.3.4 Summary of pruning SNN controller on the hardware

To summarize, pruning the SNN Agent-87 resulted in a gradual decrease in the performance of the controller similar to the behavior observed in the DNN controller. The SNN controller was at its best when no neurons were pruned and this implies that there is no or minimal redundancy of connections in the architecture of the SNN controller.

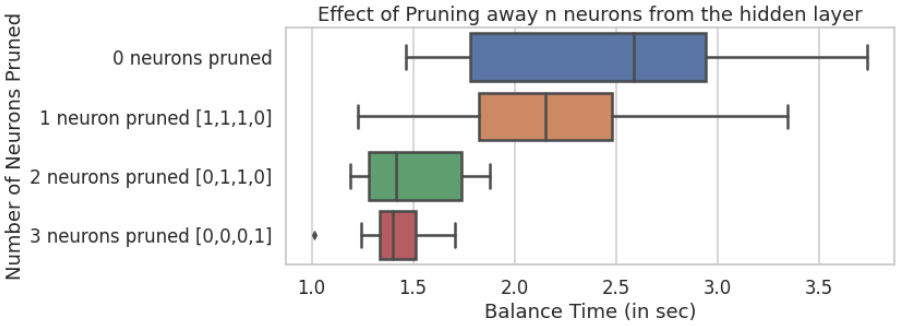


Figure 9.22: Performance of Agent-87 after pruning n neurons from the hidden layer on the hardware.

Chapter 10

Comparisons, Discussion, and Conclusions

In this chapter, we perform a comparison of the DNN and SNN agents and evaluate their energy consumptions. We define the energy models that were used to compute and compare the energy used by both the controllers in controlling the cartpole. Next, we discuss the results in brief and mention the key insights observed. We also explain the reason for the pattern of results obtained in our thesis. Lastly, we conclude on our research by revising the aim of work and reflect on the methods and results of the thesis followed by answering the research questions. We wrap-up the thesis with the future scope and recommendations in the end.

10.1 Performance vs. Complexity

We obtained Agent-45 and Agent-87 as DNN and SNN category of controllers respectively and attempted to optimize them through pruning as discussed in the previous chapter. By varying the configurations of each of the controllers, we recorded their performance in terms of average balance time. This allows us to plot a Performance vs. Complexity graph that describes how the performance of the AI controllers varies with respect to their complexity. In our case, the controller is a neural network and thus, its complexity is given by the number of parameters i.e. the number of weights and biases it requires. In the summary of pruning methodology that we discussed in chapter 9, we obtained the set of best performing DNN and SNN agents when 1, 2, and 3 neurons from their hidden layers were pruned. The best performing pruned DNN agents are given in figure 9.13 whereas the best performing SNN agents are shown in figure 9.22. With the removal of each neuron from the hidden layer, all the weights and biases associated with that neuron also become redundant. Thus, the mapping of the number of neurons pruned to the number of parameters eliminated is given in table 10.1. As discussed earlier, an unpruned DNN or SNN controller has 10 neurons in its architecture accounting to 30 parameters. As we remove neurons, the number of parameters decreases. Thus, using the results of pruning, we can plot the performance vs. complexity graph in figure 10.1.

Number of neurons pruned from the hidden layer	**Number of parameters reduced**	**Complexity of the controller (in terms of parameters)**
0	0	30
1	7	23
2	14	16
3	21	9

Table 10.1: Variation in the complexity of the controller with respect to the number of neurons pruned.

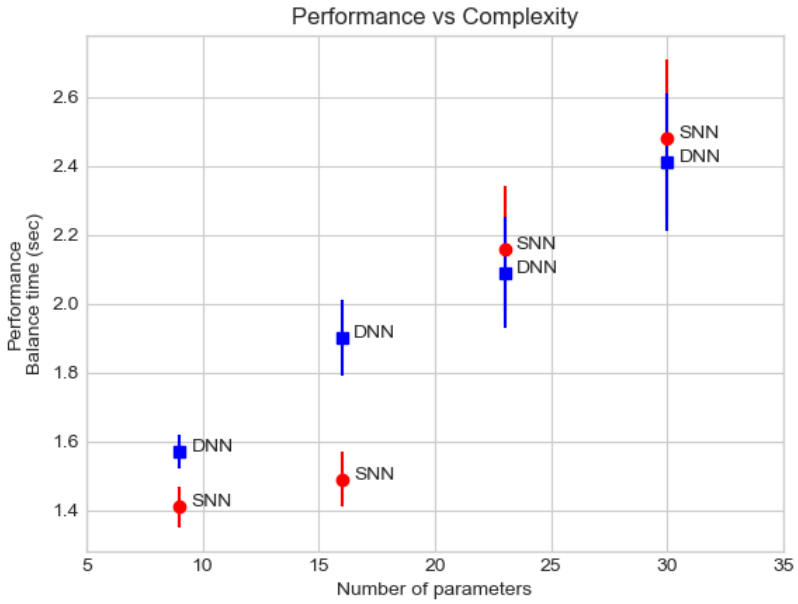


Figure 10.1: Performance versus Complexity graphs for different SNN and DNN agent configurations.

The average balance time of the unpruned Agent-87 was 2.48 seconds while that of the unpruned Agent-45 was 2.42 seconds. These agents gave the best performances as compared to their pruned versions which is evident from figure 10.1. The vertical lines for each point in the performance vs. complexity graph indicate the standard error of the mean. Using the standard error of the mean, it is observed that the performance of both the unpruned agents is nearly equal. The performance of SNN and DNN agents was nearly similar when a single neuron was pruned from the hidden layer of each of the controllers. However, when additional neurons were pruned e.g. two or three, the performance of the SNN controller dropped significantly. It must be noted that an average balance time of 1.4 seconds is higher than the time required for the free-fall of the pole initialized upright. Thus, with 3 neurons pruned in the hidden layers of both SNN and DNN agents, the agents still attempt to balance the pole but with very poor performance.

Next, we choose the best configurations of each category i.e. DNN and SNN for further consideration of energy consumptions. The pruning methodology allowed to reduce the complexity of the controllers at the cost of performance. As seen in figure 10.1, the best-case performance of the DNN and SNN controllers is 2.42 seconds and 2.48 seconds of average balance time on the hardware. Since this benchmark itself is not very high in terms of balancing time, we do not opt for any of the pruned controllers for further analysis. Thus, we choose Agent-45 and Agent-87 and profile them to examine and compare their energy consumptions.

The thesis requires that the controllers that are to be compared should provide comparable performance. It only makes sense to calculate the energy consumption of the controllers when they can provide the same level of controllability on the cartpole system. Since we chose Agent-45 and Agent-87 as our optimal controllers, we have to verify if they are providing a comparable level of performance. Despite there is a difference in the values of the average balance times of unpruned SNN and DNN agents of 0.06 seconds, we had to be sure if this difference was significant or not. To check the significance, we performed the t-test as described in the next section.

T-test

A t-test is a standard inference statistic that is used to estimate the true difference between two data populations. A t-test is used for testing a null hypothesis against the mean values of two groups. The idea behind using a t-test is to verify that the difference in the mean values of two given populations are strong and not by chance. An alternative test called the z-test also exists but it is suitable for populations of size in hundreds to thousands. In our case, we had calculated the average balance time of each of the agents after performing 30 episodes (due to time constraints) for each of the agents. Thus, we opt for doing a t-test on the mean values that we suspect to be similar. A t-test can be of three types namely an Independent two-sample t-test to compare the means between two independent groups, a Paired t-test to compare the means with the same group for different parameters, and a One-sample t-test to test the mean of a group against a known mean. Since we have two populations i.e. 30 episode performances of DNN Agent-45 and SNN Agent-87 each, we have to opt for a two-sample t-test. Let us formally state our problem to carry out the two-sample t-test.

Problem: We assume that there are no differences in the average balance time of Agent-45 and Agent-87. We would like to know if the difference between the average balance time of DNN and SNN controllers is significant or not. The data that we have for performing the test is given in table 10.2.

Balance Time	**SNN**	**DNN**
Mean	2.48	2.42
Standard Deviation	0.88764	0.81503
Number of samples	30	30

Table 10.2: Statistics about the SNN and DNN performances

Solution: To begin with, we first construct our null hypothesis H_0:$\mu_{DNN} = \mu_{SNN}$, meaning that the average of balance time of DNN and SNN are the same. Next, we construct the alternative hypothesis H_a:$\mu_{DNN} \neq \mu_{SNN}$, meaning that we want to check if the mean values between the two controllers differ from each other or not. Thus, using the values given in table 10.2, we calculate the t-statistic using the formula,

$$t = \frac{\text{sample difference} - \text{hypothesized difference}}{\text{Standard error of the difference}}$$

$$t = \frac{(\bar{x}_{SNN} - \bar{x}_{DNN}) - (\bar{\mu}_{SNN} - \bar{\mu}_{DNN})}{\sqrt{\frac{s^2_{SNN}}{n_{SNN}} + \frac{s^2_{DNN}}{n_{DNN}}}}$$

where μ_{SNN} and μ_{DNN} are the mean values considering the null hypothesis. Since our null hypothesis says there is no difference in the mean values, the difference is zero. Furthermore,

$\bar{x}_{SNN}$ and $\bar{x}_{DNN}$ represent the sample means of SNN and DNN respectively

s_{SNN} and s_{DNN} represent the standard deviations of SNN and DNN respectively, and

n_{SNN} and n_{DNN} denote the sample size of SNN and DNN groups respectively.

Substituting these values to compute the value of t,

$$
\begin{aligned}
t &= \frac{(2.48 - 2.42) - (0)}{\sqrt{\frac{0.8876^2}{30} - \frac{0.8150^2}{30}}} \\
&= \frac{0.06}{\sqrt{\frac{0.7878 + 0.6642}{30}}} \\
&= \frac{0.06}{\sqrt{0.0484}} \\
&= \frac{0.06}{0.22}
\end{aligned}
$$

$$t = 0.2727$$

Once we have calculated the t-value, as a next step, we have to calculate the p-value. A p-value indicates the probability with which the mean values were obtained by chance. To exemplify, if we calculated a p-value = 0.05, it means that the average values that we have obtained regarding the SNN and DNN were not by chance. The lower the value of p, the higher is the reliability of the results. Usually, a p-value of 0.05 i.e. 5% or less is considered to validate the averages of the two datasets.

To use the t-table, we require another variable i.e. degree of freedom (df). A df is directly related to the size of the population. The df, in the case where the two samples sets are of the same size, is calculated as,

$$
\begin{aligned}
df &= 2n - 1 \\
df &= (2 \times 30) - 1 \\
df &= 59
\end{aligned}
$$

The problem we are considering is a 'two-tail' problem since our alternative hypothesis is H_a:$\mu_{DNN} \neq \mu_{SNN}$. Thus, we have to look up in the t-table (can be found in the Appendix C) across the row with df = 59 for the values near our test statistic $t = 0.2727$. A snippet of the t-table is shown in figure 10.2. We observe that we do not have an exact match for our value of df=59. In such cases, we use the closest row below df=59 i.e $df = 40$.

We can see that the p-value corresponding to our t-value for a two-tails test lies in between 1.00 and 0.50 i.e. between 100% and 50%. We can also calculate the p-value using technology (i.e. online) using the t-values. The p-value turned out to be 0.78943 which is way larger than the allowed p-value of 0.05.

As a result, we fail to reject our null hypothesis i.e. $\mu_{DNN} = \mu_{SNN}$, and thus this hypothesis still holds. Therefore, we conclude that the average balance time of 2.48 seconds for the SNN controller and 2.42 seconds for the DNN controller are not significantly different. With this result, we have a pair of AI controllers: unpruned Agent-45 and unpruned Agent-87 with comparable performance on the real cartpole system.

cum. prob	$t_{.50}$	$t_{.75}$	$t_{.80}$	$t_{.85}$	$t_{.90}$
one-tail	0.50	0.25	0.20	0.15	0.10
two-tails	1.00	0.50	0.40	0.30	0.20
df					
1	0.000	1.000	1.376	1.963	3.078
2	0.000	0.816	1.061	1.386	1.886
3	0.000	0.765	0.978	1.250	1.638
4	0.000	0.741	0.941	1.190	1.533
⋮	⋮	⋮	⋮	⋮	⋮
22	0.000	0.686	0.858	1.061	1.321
23	0.000	0.685	0.858	1.060	1.319
24	0.000	0.685	0.857	1.059	1.318
25	0.000	0.684	0.856	1.058	1.316
26	0.000	0.684	0.856	1.058	1.315
27	0.000	0.684	0.855	1.057	1.314
28	0.000	0.683	0.855	1.056	1.313
29	0.000	0.683	0.854	1.055	1.311
30	0.000	0.683	0.854	1.055	1.310
40	0.000	0.681	0.851	1.050	1.303
60	0.000	0.679	0.848	1.045	1.296
80	0.000	0.678	0.846	1.043	1.292
100	0.000	0.677	0.845	1.042	1.290
1000	0.000	0.675	0.842	1.037	1.282

Figure 10.2: Extracting the p-value using t-value and df from the t-table.

10.2 Memory Consumption

The agents trained to control the cartpole system are designed in Python. One of the key features of Python is that it ensures the memory management internally using the 'python memory manager' and that the user has no control over it. Despite this feature makes the manual calculation of memory usage by the agents complicated and trivial, it cannot be ignored. In this section, we calculate the memory consumptions of each agent in terms of different sections of the python script and compare them.

We used Python 3.4 version for scripting that includes a debug tool called 'tracemalloc' [2]. This tool is used to trace the size of memory blocks allocated by the python script executing the agent. The script to run each of the agents has the structure as shown in listing 10.1.

```
# Section 1: IMPORTING LIBRARIES

# Section 2: SETTING PARAMETERS (Weights and Biases)

# Section 3: DEFINING FUNCTIONS

# Section 4: SETTING UP THE STATE SPACE (2D array)

steps = 500

for i in range(steps):

    # Section 5: FEED-FORWARDING
```

Listing 10.1: Structure of the python script used to run the agent

As seen from above, the script mainly comprises five sections namely: importing libraries, setting parameters of the network, defining functions, setting a dummy state-space, and finally actual execution of the network through the feed-forward process. Speaking of memory consumption of the AI controller, we are mainly concerned here with the amount of memory taken by the network parameters, functions that are required to produce the control actions, and the feed-forwarding of the network. The remaining sections in the listing 10.1 are common for both the agents and would also take up the same amount of memory space.

10.2.1 Memory consumptions for one step

We implemented the script given in the listing 10.1 on the laptop to calculate the memory consumption. The test was not carried out on the hardware to avoid the interference of memory usage due to sensor and motors. A dummy state space was fed to the network and the memory consumption was traced using tracemalloc.

As seen from figure 10.3, we see a higher memory footprint by SNN agent as compared to the DNN agent while setting parameters and actual execution of the network per step. By per step, we mean that the network executes feed-forward once and calculates one action for a state fed into the network.

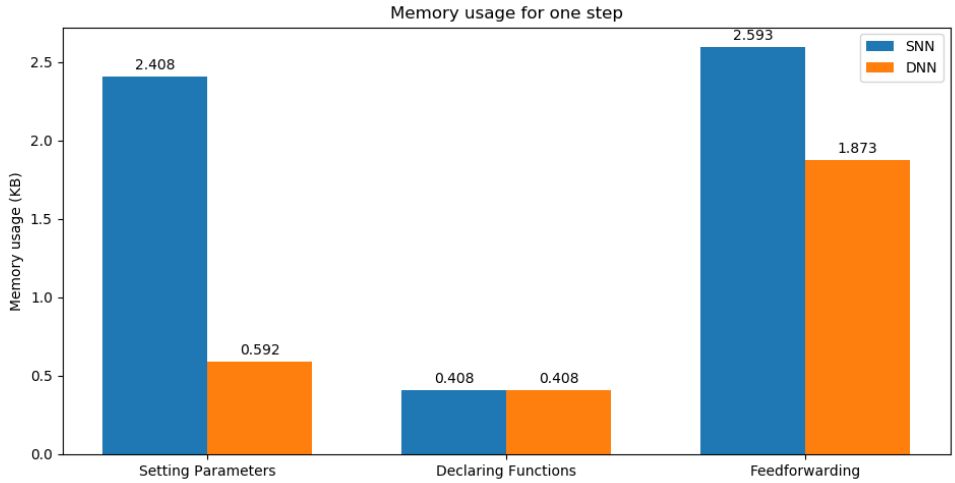


Figure 10.3: Memory footprint of one step execution by each agent.

Further, we see a substantially higher memory used by SNN in setting the parameters despite the number of neurons in both the agents is the same i.e 10. Apart from setting the weights and biases of the SNN agent, it requires initializing a few additional variables to calculate its internal state dynamics which is the cause for a higher memory footprint as compared to the DNN agent. Both SNN and DNN agents display the same amount of memory allocation for declaring functions because memory is not allocated to the functions until their execution.

Finally, unlike DNN, the SNN performs more number of calculations owing to its internal state dynamics associated with every spiking neuron. Thus, except the input layer, SNN keeps a track of the membrane potential of all the neurons in the hidden layer and the output layer to decide when that particular neuron will fire i.e. produce a spike. Thus, there are additional variables required to keep a check of each neuron's membrane potential. This makes an SNN costlier in terms of the memory footprint as compared to DNN.

10.3 Energy Consumption

In our project, one of the key differences in the control strategies of SNN and DNN agents is the sampling frequencies of each agent. Despite the same architectures of both the controllers, the SNN agent has a sampling time of 35 ms owing to its methodology, whereas the DNN agent has a sampling time of 25 ms. Thus, theoretically, the SNN agent requires approximately 28 correct[1] actions to balance the pole for 1 second. On the other hand, DNN would require 40 correct actions

[1]It implies that the action taken by the agent did not result in the failure of the pole balancing task.

to balance the pole for 1 second.

Both the SNN and DNN agents have an equal number of neurons i.e. 10 (4 input, 4 hidden, and 2 output). However, SNN has additional parameters inherently to keep track of the internal states (commonly referred to as internal state dynamics) of its neurons. Therefore, in each feedforward mechanism, these parameters play a vital role and are updated for every neuron in the network. As a result, apart from the basic feedforward calculations done by the DNN, SNN performs additional calculations owing to the internal state dynamics of the neurons leading to an overhead.

In summary, DNN holds an advantage over SNN in terms of fewer calculations per action whereas SNN has an upper hand when we consider the number of actions required to balance the pole over a longer period. In this section, we scrutinize the energy consumption of each of the agent.

Strategy for calculating energy consumptions of AI controllers

EV3 has very simplistic hardware when it comes to interface. It has four input ports, four output ports, a mini-USB PC Port for connecting to a computer, and the power supply port. To measure the power consumption of a process, the best option is to take the readings from the actual hardware in some way. These measurements indicate the power consumption of the running programs as well as other things such as the brightness of the monitor (in case of a laptop). Thus, the measurements are termed as global or whole-system measurements. We can use ammeters directly on the hardware to read the power numbers, but in case of EV3, this is not possible as there is no direct access to the hardware or availability of ports that can be used to measure power. Another method to measure the energy consumptions of a program is by measuring the difference in the battery drainage when the programs are run for a longer duration of time. However, this would simply give an unclassified single value of energy consumption of the EV3 as a whole. The difference in the battery drainage would also include the energy used by the motors and sensors that are used for the application. Thus, the approach of tracking battery drainage to estimate the energy consumption of the AI controllers is not a good option. In summary, measuring the energy consumptions on the hardware is difficult.

We chose an approach of simulating the AI controllers on a laptop to measure their internal energy consumption. The idea is to have a ratio of energy consumptions of DNN controllers to SNN controllers, and conclude with the most efficient controller. The laptop we used is equipped with a powerful Intel processor as compared to the simple ARM9 processor used in the EV3 controller. The advantage of simulating the AI controllers on a laptop is the possibility of being able to measure the internal energy consumptions. Latest Intel processors implement the RAPL (Running Average Power Limit) interface that provides model-specific registers (MSRs) containing energy consumption estimates for up to four power planes or domains of a machine. These MSRs are unavailable on the LEGO EV3 controller which limit the energy profiling to be carried out on the real cartpole system. A processor has four power planes implies that there are four different voltages required by the processor to power its core, I/O pins, and other hardware components. The power planes can be visualized in figure 10.4. A processor has one or more packages. Client processors, e.g. Core i3, i5, and i7 have a single package. Each package contains multiple cores. A package can be broadly classified into two parts, the 'Core' and the 'Uncore'. The core typically comprises computational units like ALU and FPU whereas the uncore consists of the last level caches (LLCs) and the memory controller. The core, the package, and the graphics (i.e. GPU) energy estimates can be directly obtained through the MSRs. To obtain the energy consumption due to the caches, we can subtract the energy consumptions of core and graphics from the package energy. The Intel processors calculate the energy estimates through a power model that uses processor-internal counts as inputs. The energy numbers obtained using this process have been proved in [11] to be fairly accurate and reliable since they are updated at a frequency of 1000 Hz. As the next step, we have to use a tool to extract the energy estimates via the RAPL interface.

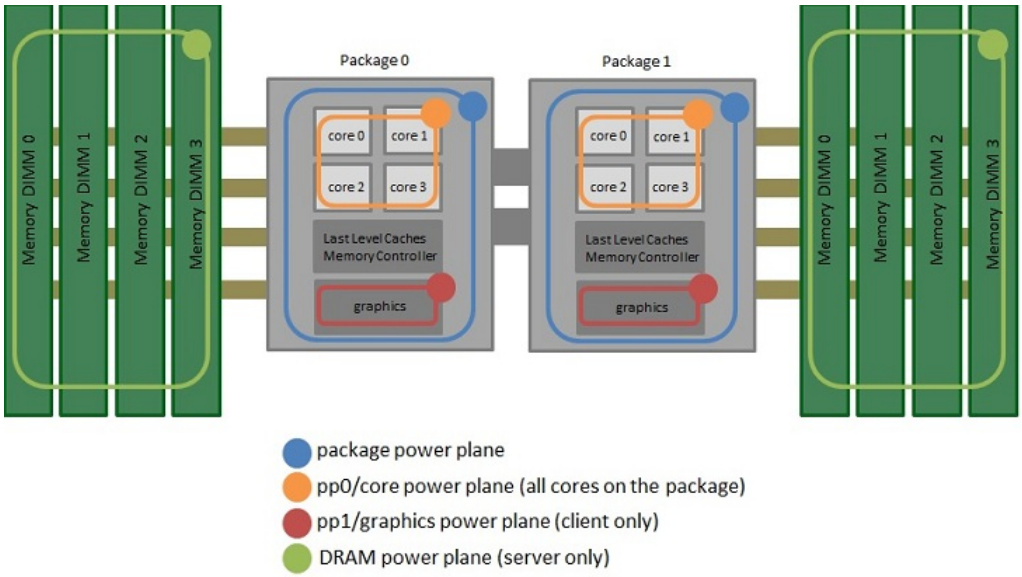


Figure 10.4: Power planes in the Intel processor taken from [7].

Perf as the Energy Profiler tool

Some of the commonly used tools to read the RAPL estimates are Perf (Linux based), Intel Power Gadget (Linux, MAC, and Windows), powermetrics (Mac), and turbostat (Linux). Among these options, we choose the Perf tool as the energy profiler tool because it is simple, flexible, and used via the command-line interface. It is a powerful system-wide instrumentation service for Linux based systems. It is capable of printing all the available Intel RAPL power estimates at specified regular intervals. The only care that had to be taken while taking is the measurements was that, no other activities apart from the controller-program should be running on the system. This was essential since the RAPL power estimates are system-wide estimates. Next, we had to install and use the perf tool on the laptop. The key requirement for using perf tool was that the laptop should be running Linux OS. However, we had a Windows-based laptop. Therefore, we had the following two options to use the perf tool on our laptop:

1. Installing a Linux Virtual Machine on Windows, and
2. Using Linux via creating a bootable Ubuntu USB stick on Windows.

We initially tried to install Linux on the Virtualbox on the Windows laptop. Despite the installation was successful, the readings that we obtained were incomplete. Various metrics such as the number of cycles, instructions, branches, and energy estimates of cores, package, etc. were not supported. On investigation, we understood that the perf tool does not work in the Virtualbox environment as Windows does not support APIs that allow easy access to the MSRs. The perf tool only works on a local machine running Linux. As a result, we moved to the second option of creating a bootable Ubuntu USB stick on Windows. With the bootable flash drive, we could use the perf tool to its full extent. The processor specifications of the laptop that we used to simulate and profile the AI controllers are given in table 10.3.

10.3.1 Energy consumption per step

Procedure: To describe the process that we used to calculate the energy consumptions, consider the structure of the python script that was used to execute the AI controllers in listing 10.1. It mainly consists of 5 sections namely, importing libraries, setting parameters, defining functions,

Processor	Intel(R) Core(TM) i7-7700HQ CPU @ 2.80 GHz
Cores	4
L1 Cache	256KB
L2 Cache	1 MB
L3 Cache	6 MB

Table 10.3: Specifications of the system used for profiling AI controllers for calculating energy and memory consumptions.

setting up the state space, and feed-forwarding. A key aspect in understanding the script is that the profiling of the AI controllers had to be done on the simulations, i.e. the controllers would not be able to directly interact with the real plant. We already know that both the controllers are capable of attaining an average balance time of 2.42 seconds and 2.48 seconds. These controllers are also capable of balancing the simulated cartpole system i.e. Cartpole-v1 environment from the OpenAI gym toolkit substituted with the real hardware parameters. However, to simulate the AI controllers on the laptop, we used a dummy state space instead of OpenAI Gym's cartpole environment. We avoided using the OpenAI Gym's environment to remove the overhead in calling the library. The dummy state space was fed into the network, and the feedforwarding action was monitored. In summary, the script given in listing 10.1 was used to simulate the AI controllers on the laptop over a dummy state space that represented a cartpole system.

Moving further, the energy consumption that we are focussing on is the energy used in performing the feed-forward calculations since feed-forwarding is mainly responsible to produce the controlling actions. The process of obtaining the energy of the feedforwarding section from the python script was fivefold:

1. Import the NumPy library, set the weights and biases of the controller, and produce a dummy state-space of the cartpole system with state variables in the ranges [$x \in$ [-2.5, 2.5], $\dot{x} \in (-\infty, \infty)$, $\theta \in$ [-0.61, 0.61], $\dot{\theta} \in (-\infty, \infty)$] with number of entries $\geq$ step variable.

2. Set step = 500. Execute the python script using the perf tool to obtain the energy numbers. Run the script 100 times and calculate the average of the energy values (to get accurate and reliable results with less deviation). Note the values for core energy and package energy. Calculate the uncore value by subtracting core energy from package energy.

3. Now set step = 0 and repeat the process as done in step 2.

4. Subtract the values obtained in step 3 from step 2 to obtain the energy relevant to the feedforward process executed 500 times. This gives the energy for performing feedforwarding process 500 times i.e 500 steps.

5. Divide each result by 500 to obtain the energy per step for the controller.

Table 10.4 demonstrates the steps 2 until 5 by using variables for core, uncore, and package. The exact process is repeated for the SNN controller and the readings for both the controllers are recorded.

	STEPS	CORE	PKG	UNCORE
Step 2	500	Core_{500}	Pkg_{500}	$\text{Uncore}_{500} = \text{Pkg}_{500}$ - Core_{500}
Step 3	0	Core_{0}	Pkg_{0}	$\text{Uncore}_{0} = \text{Pkg}_{0} - \text{Core}_{0}$
Step 4	Difference	$\text{Core}_{\text{diff}} = \text{Core}_{500}$ - Core_{0}	$\text{Pkg}_{\text{diff}} = \text{Pkg}_{500}$ - Pkg_{0}	$\text{Uncore}_{\text{diff}} = \text{Uncore}_{500}$ - Uncore_{0}
Step 5	Energy /step	$\text{Core}_{\text{step}} = \text{Core}_{\text{diff}}$ / 500	$\text{Pkg}_{\text{step}} = \text{Pkg}_{\text{diff}}$ / 500	$\text{Uncore}_{\text{step}} = \text{Uncore}_{\text{diff}}$ / 500

Table 10.4: Illustration of stepwise calculation of core, uncore, and package energy per step for SNN and DNN controllers.

Results: As discussed earlier, we know that SNN performs more number of calculations to produce a single action as compared to DNN. Based on this theory, it is expected that the SNN agent would also require more energy than DNN per step. As seen from figure 10.5, the SNN agent required almost twice the energy required for computation in the cores and the uncore planes than the DNN. In summary, referring to the sum of core and uncore power planes as indicated by the package energy bar, it can be concluded that SNN consumes nearly 1.7 times the energy required by DNN to perform a single feedforward step calculation.

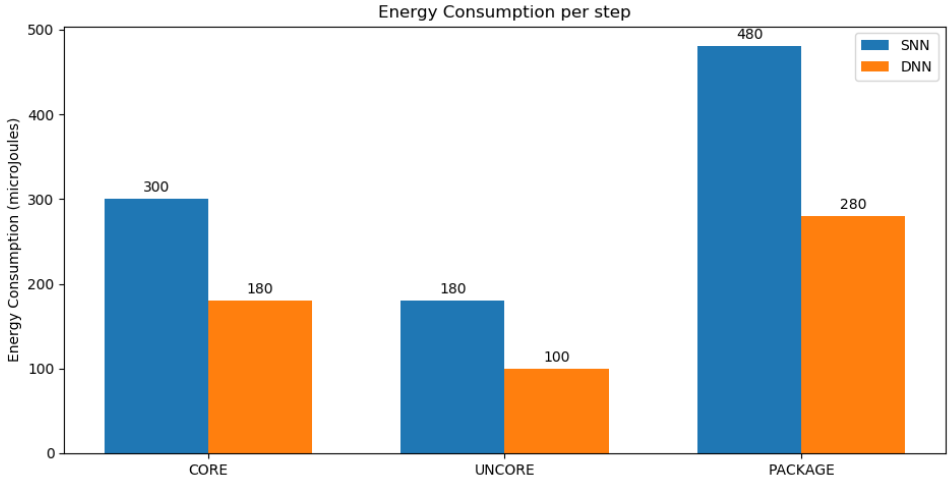


Figure 10.5: Energy consumption of one step execution by each agent.

10.3.2 Energy consumption per second

By energy consumption per second, we mean the energy required by the agents to balance the pole for a duration of 1 second. Considering the sampling times of SNN and DNN agents as 35 ms and 25 ms, we calculate that they require to compute 28.57 and 40 actions reapectively. Since, the number of actions can only be an integer, we calculate a balance time common for both the agents wherein they can take integral number of actions. We choose a balance time of 35 seconds for which SNN agent would require 1000 steps (35/0.035) whereas DNN agent would require 1400 steps (35/0.025). Once again, the steps are followed similar to the previous case with minor modifications as illustrated in the table 10.5 and table 10.6 for DNN and SNN controllers respectively. To obtain the energy per second, we divide the results obtained in step 4 by 35 in both the cases.

	STEPS	CORE	PKG	UNCORE
Step 2	1400	Core_{1400}	Pkg_{1400}	$\text{Uncore}_{1400} = \text{Pkg}_{1400}$ - Core_{1400}
Step 3	0	Core_{0}	Pkg_{0}	$\text{Uncore}_{0} = \text{Pkg}_{0}$ - Core_{0}
Step 4	Difference	$\text{Core}_{\text{diff}} = \text{Core}_{1400}$ - Core_{0}	$\text{Pkg}_{\text{diff}} = \text{Pkg}_{1400}$ - Pkg_{0}	$\text{Uncore}_{\text{diff}} = \text{Uncore}_{1400}$ - Uncore_{0}
Step 5	Energy /step	$\text{Core}_{\text{step}} = \text{Core}_{\text{diff}}$ / 35	$\text{Pkg}_{\text{step}} = \text{Pkg}_{\text{diff}}$ / 35	$\text{Uncore}_{\text{step}} = \text{Uncore}_{\text{diff}}$ / 35

Table 10.5: Illustration of stepwise calculation of energy consumption per second for DNN controller.

From figure 10.6, it is evident that SNN's advantage of having a higher sampling time does not help it overcome the energy costs as compared to DNN. The overall energy consumption of SNN for balancing the pole for 1 second is 14000 µJoules whereas DNN consumes 10285 µJoules. The same pattern is observed in the core and uncore planes where SNN is consuming higher energy as compared to DNN. In conclusion, the SNN controller required approximately 1.4 times the energy of the DNN controller to balance the pole for one second.

	STEPS	CORE	PKG	UNCORE
Step 2	1000	Core_{1000}	Pkg_{1000}	$\text{Uncore}_{1000} = \text{Pkg}_{1000}$ - Core_{1000}
Step 3	0	Core_{0}	Pkg_{0}	$\text{Uncore}_{0} = \text{Pkg}_{0} - \text{Core}_{0}$
Step 4	Difference	$\text{Core}_{\text{diff}} = \text{Core}_{1000}$ - Core_{0}	$\text{Pkg}_{\text{diff}} = \text{Pkg}_{1000}$ - Pkg_{0}	$\text{Uncore}_{\text{diff}} = \text{Uncore}_{1000}$ - Uncore_{0}
Step 5	Energy /step	$\text{Core}_{\text{step}} = \text{Core}_{\text{diff}}$ / 35	$\text{Pkg}_{\text{step}} = \text{Pkg}_{\text{diff}}$ / 35	$\text{Uncore}_{\text{step}} = \text{Uncore}_{\text{diff}}$ / 35

Table 10.6: Illustration of stepwise calculation of energy consumption per second for SNN controller.

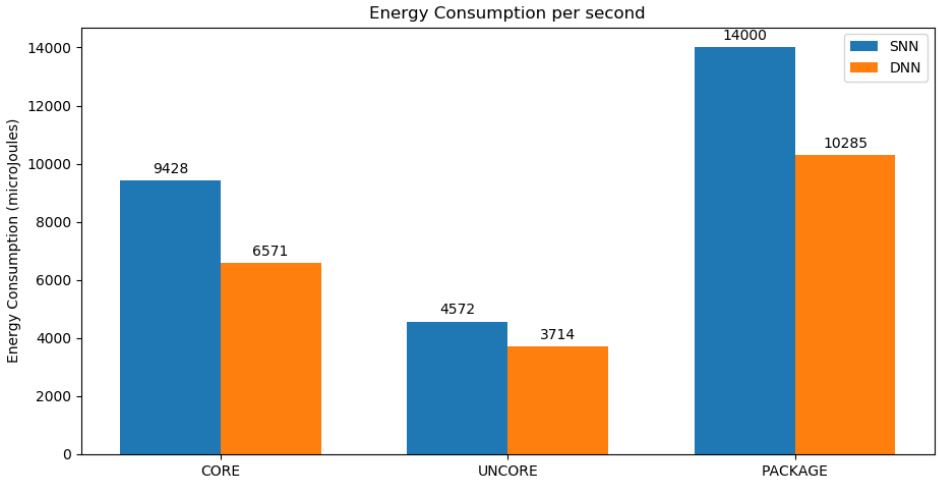


Figure 10.6: Energy consumption for balancing the pole for one second by each agent.

10.4 Energy Consumptions based on Multiply-Accumulate (MAC) operation

Spiking Neural Networks, at the heart, are driven by a sparse event-based mechanism. Precisely, the spikes that propagate through the network are responsible for the communication and computation in SNNs. Mathematically, these spikes are binary values (0 or 1). SNNs are popularly known to be energy-efficient solutions due to their inherent nature of event-driven hardware operation. In the following section, we evaluate the energy efficiency of SNN and DNN agent with respect to the Multiply-Accumulate (MAC) operations.

A Multiply-Accumulate (MAC) operation is a commonly used term in computation which calculates the product of two numbers and adds the result to an accumulator 'a' as shown below:

$$a \Leftarrow a + (b \times c)$$

Generally, there are dedicated hardware units equipped in the computers that perform this type of operation. These are called a MAC unit or simply MAC. To understand how the MAC operations play a role in the neural network, consider the example given in figure 10.7.

For a standard artificial neuron, a single MAC operation is required per input as seen from figure 10.7 (upper half). As we increase the number of inputs to the neuron, the number of MAC operations increases proportionally. As seen from figure 10.7 (lower half), with three inputs to the neuron, we require to do three MAC operations. Thus, we can say that the number of MAC operations that a neuron requires is equal to the number of inputs it is connected to.

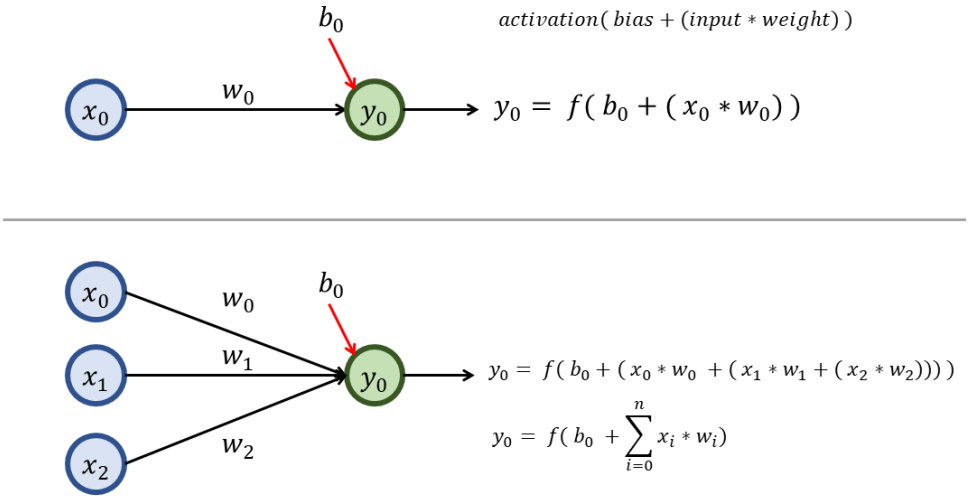


Figure 10.7: An example of a neuron with inputs x_i, their corresponding weights w_i and bias b_i. The top network depicts the output of a single input to a neuron. The bottom network depicts the output of three inputs to a neuron.

10.4.1 Energy Model for DNN

Based on the theoretical concepts described for the MAC operations in a neural network, we build an energy model based on the number of MAC operations for a DNN as,

For a layer l with m inputs and n neurons,

$$\text{Energy}_l = m \times n \times \text{Energy}_{\text{MAC}} \tag{10.1}$$

where,
Energy_l = Energy per layer
$\text{Energy}_{\text{MAC}}$ = Energy required for a single MAC operation.

The structure of our DNN Agent-45 consists of 4 inputs, 4 neurons in the hidden layer and 2 neurons at the output. Therefore, we calculate the energy consumption of the DNN agent using equation 10.1 as:

$$\begin{aligned}
\text{Energy}_{\text{i/p}} &= 0 \times 4 \times \text{Energy}_{\text{MAC}} = 0 \times \text{Energy}_{\text{MAC}} \\
\text{Energy}_{\text{hid}} &= 4 \times 4 \times \text{Energy}_{\text{MAC}} = 16 \times \text{Energy}_{\text{MAC}} \\
\text{Energy}_{\text{o/p}} &= 2 \times 4 \times \text{Energy}_{\text{MAC}} = 8 \times \text{Energy}_{\text{MAC}}
\end{aligned}$$

$$\text{Energy}_{\text{DNN}} = \text{Energy}_{\text{i/p}} + \text{Energy}_{\text{hid}} + \text{Energy}_{\text{o/p}} = \mathbf{24} \times \mathbf{Energy_{MAC}}$$

10.4.2 Energy Model for SNN

Since we are implementing the SNN controller on a non-neuromorphic platform, spiking neurons perform similar computations as that of the Artificial neurons except that the neurons do not fire at every step. We, therefore, consider the average firing rate of neurons to compute the total

energy in terms of MAC operations. Apart from the basic feedforward functionality, the network also keeps track of the internal dynamics of each neuron. We build an energy model as given in [28] based on number of MAC operations for an SNN as:

For a layer l with m inputs and n outputs,

$$\text{Energy}_l = (mn \ + \ 2nn) \ \times \ \text{F}_r \ \times \ \text{Energy}_{\text{MAC}} \tag{10.2}$$

where,
Energy_l = Energy per layer
F_r = Firing rate of neuron
$\text{Energy}_{\text{MAC}}$ = Energy required for a single MAC operation.

The structure of our SNN agent is similar to that of the DNN agent and it consists of 4 inputs, 4 neurons in the hidden layer, and 2 neurons at the output. In our experiments, we calculated the firing rate $\text{F}_r = 0.17$. It is calculated by running the simulation for certain steps denoted by s and summing up the total number of spikes generated denoted by $total_{spikes}$. Finally, the firing rate is calculated as $(total_{spikes} \ / \ s) \ / \ n$ where n is the total number of neurons that fired. We calculate the energy consumption of the SNN agent substituting the values in equation 10.2 as,

$$\begin{aligned}
\text{Energy}_{\text{i/p}} &= \text{NA (as there is no internal state calculation)} \\
\text{Energy}_{\text{hid}} &= (16 + 2 \times 16) \times \ \text{F}_r \times \text{Energy}_{\text{MAC}} = 8.16 \times \text{Energy}_{\text{MAC}} \\
\text{Energy}_{\text{o/p}} &= (\ 8 \ + 2 \times 4 \ ) \times \ \text{F}_r \times \text{Energy}_{\text{MAC}} = 2.72 \times \text{Energy}_{\text{MAC}}
\end{aligned}$$

$$\text{Energy}_{\text{SNN}} = \text{Energy}_{\text{i/p}} + \text{Energy}_{\text{hid}} + \text{Energy}_{\text{o/p}} = \mathbf{10.88 \times Energy_{MAC}}$$

From the above results, it is conclusive that the SNN is 0.45 times expensive than DNNs in terms of MAC operations. Unlike general processors, Neuromorphic processors are more efficient in training and running spiking neural networks. Spikes are considered as binary values in neuromorphic computers that act as a switch to enable or disable a multiply operation. Hence, a MAC operation for an SNN on a Neuromorphic platform acts as an accumulate (AC) operation and the energy model given by equation 10.2 is modified as:

$$\text{Energy}_l = mn \ \times \ \text{F}_r \ \times \ \text{Energy}_{\text{AC}} \tag{10.3}$$

$$\text{Energy}_l = mn \ \times 0.17 \ \times \ \text{Energy}_{\text{AC}} \tag{10.4}$$

Calculating the energy layer wise, we get,

$$\begin{aligned}
\text{Energy}_{\text{i/p}} &= \text{NA (as there is no internal state calculation)} \\
\text{Energy}_{\text{hid}} &= (4 \times 4) \times 0.17 \times \text{Energy}_{\text{AC}} = 2.72 \times \text{Energy}_{\text{AC}} \\
\text{Energy}_{\text{o/p}} &= (4 \times 2) \times 0.17 \times \text{Energy}_{\text{AC}} = 1.36 \times \text{Energy}_{\text{AC}}
\end{aligned}$$

$$\text{Energy\_Neuromorphic}_{\text{SNN}} = \text{Energy}_{\text{i/p}} + \text{Energy}_{\text{hid}} + \text{Energy}_{\text{o/p}} = \mathbf{4.08 \times Energy_{AC}}$$

A 32-bit MAC Int requires 3.2pJ and a 32-bit AC Int require 0.1pJ [17] which would make running SNNs running on a Neuromorphic platform much faster. Thus, the thesis also provides a scope and motivation for further development of neuromorphic platforms for SNN implementation to obtain energy efficient controllers. We provide a brief comparison on the energy per layer in terms of MAC operations between ANN and SNN in table 10.7.

Platform	Energy per layer in terms of MAC operations	
	ANN	**SNN**
General processor	$m \times n \times E_{MAC}$	$(mn + 2nn) \times E_{MAC} \times F_r$
Neuromorphic	Not Supported	$mn \times E_{AC}$

Table 10.7: Energy consumption per layer for ANN and SNN on the different target platforms. m and n denote the number of inputs to a neuron and total number of neurons in a layer respectively. F_r denotes the firing rate of the spiking neurons. E_{MAC} and E_{AC} denote the energy required for a MAC and an AC operation respectively.

10.5 Energy Comparison Summary

We built two energy models to compute and compare the energy consumption of AI controllers. The first model was a practical calculation taken by profiling the AI controllers on the laptop. The second energy model was a theoretical calculation based on the number of MAC operations. In this section, we summarize the methodology and the results obtained for a clear comparison.

The energy models that were built practically and theoretically are stated in table 10.8. For the practical approach, the term Pkg_n denotes the package energy when the controller produced 'n' number of total actions. In the theoretical approach, m_l and n_l denote the number of inputs to a neuron and the total number of neurons in the layer l respectively. E_{MAC} and E_{AC} denote the energy required fora MAC and an AC operation respectively. F_r denotes the firing rate of the spiking neurons. Note that we take the sum of energy for three layers ($l = 3$) of our controllers in the theoretical approach.

Approach	Energy Models			
	DNN		SNN	
Practical	**Per step**	**Per second**	**Per step**	**Per second**
	$(Pkg_{500} - Pkg_0)/500$	$(Pkg_{1400} - Pkg_0)/35$	$(Pkg_{500} - Pkg_0)/500$	$(Pkg_{1000} - Pkg_0)/35$
Theoretical	**Gen. processor (MAC)**	**Neuromorphic (AC)**	**Gen. processor (MAC)**	**Neuromorphic (AC)**
	$\sum_{l=1}^{3} m_l \times n_l \times E_{MAC}$	NA	$\sum_{l=1}^{3} (m_l n_l + 2n_l n_l) \times F_r \times E_{MAC}$	$\sum_{l=1}^{3} m_l n_l \times E_{AC}$

Table 10.8: Summary of energy comparison methodology for DNN and SNN controllers

Table 10.9 describes the results of the energy comparisons of DNN and SNN controllers. It is evident that the DNN controller is energy efficient as compared to the SNN controller when they are executed on a general-purpose processor. However, SNN dominates the energy-efficiency when we use a neuromorphic processor to run the SNN controller.

Approach	Energy Consumption			
	DNN		SNN	
Practical	**Per step**	**Per second**	**Per step**	**Per second**
Energy (μJ)	480	14000	280	10285
Theoretical	**Gen. processor (MAC)**	**Neuromorphic (AC)**	**Gen. processor (MAC)**	**Neuromorphic (AC)**
Energy (pJ)	76.8	-	34.816	0.408

Table 10.9: Summary of energy comparison results for DNN and SNN controllers

10.6 Discussion

The study demonstrates a comparison of two categories of AI controllers that are designed to control a practical continuous-time cartpole system. The process started with designing a real

cartpole system using LEGO Educational Kit. It involved hardware specifications, hardware design and assembly, software design, AI controller's study, design, implementation, testing, and finally profiling. The two categories of AI controllers considered for the thesis are Deep Neural Networks and Spiking Neural Networks. The design of DNN controllers began with implementing DQN in an offline fashion and a controller was obtained. With several modifications and multiple types of configurations, the DQN controller was able to balance the pole on the real cartpole system for an average balance time of 2.2 seconds. The main drawback of this controller was that it had to make the use of the approximate model of the cartpole system to be able to execute control over the hardware. To obtain a better version of the DNN controller than the DQN, we studied, designed, and implemented the second reinforcement learning algorithm, NFQ that was proven in [20] to be capable of solving an inverted pendulum task (similar to a cartpole problem). Owing to the difference in problem setup and the hardware limitations between the experiments done in [20] and our thesis, the parameters of the NFQ used in this thesis did not exactly match. However, the most of the settings of the network hyperparameters were nearly similar to those used in the paper [20]. Nevertheless, the NFQ agent was able to execute a performance with an average balance time of 2.2 seconds as a model-free controller. Despite it was an improvement in terms of being model-free, the performance was still not very significant. Lastly, with an aim to improve the NFQ controller's performance we used an Evolutionary algorithm called the Genetic Algorithm. Despite GA being an optimization technique used to optimize existing suboptimal solutions, it could not remarkably improve the performance of the NFQ agent. Instead, the GA algorithm was able to produce the best DNN controller when it was provided with completely untrained networks. The GA controller, Agent-45 representing the DNN category of controllers exhibited an average balance time of 2.5 seconds over the real cartpole system. Since GA showed the best capability of producing controllers among our chosen algorithms, we used it to produce the SNN category of controllers as well. The result was an SNN Agent-87 that could balance the real cartpole system with an average balance time of 2.8 seconds. These agents were later pruned to check if there existed any redundant weights or biases. With the pruning results, the Agents 45 and 87 were proved to be performing better when used unpruned. These agents were finally profiled for energy consumptions and the results were compared.

One of the major points of concern in this thesis is the notably lower benchmark set by all the algorithms used to control the practical cartpole system. Thus, we discuss the cause of this behavior. We had encountered a few issues with the hardware and also found out their solutions. The issues encountered and their countermeasures that we devised are presented below.

1. **Delays:** There exists a reality gap that disallows using simulated AI controllers directly on the real hardware. This is due to the inherent presence of delays in the hardware due to the sensors, actuators, and the controller (i.e. computation within the network) itself. To resolve this, we implemented strategies such as 'learning the delay' and 'countering the delay'. Eventually, the DQN controller was able to execute its control using the latter technique.

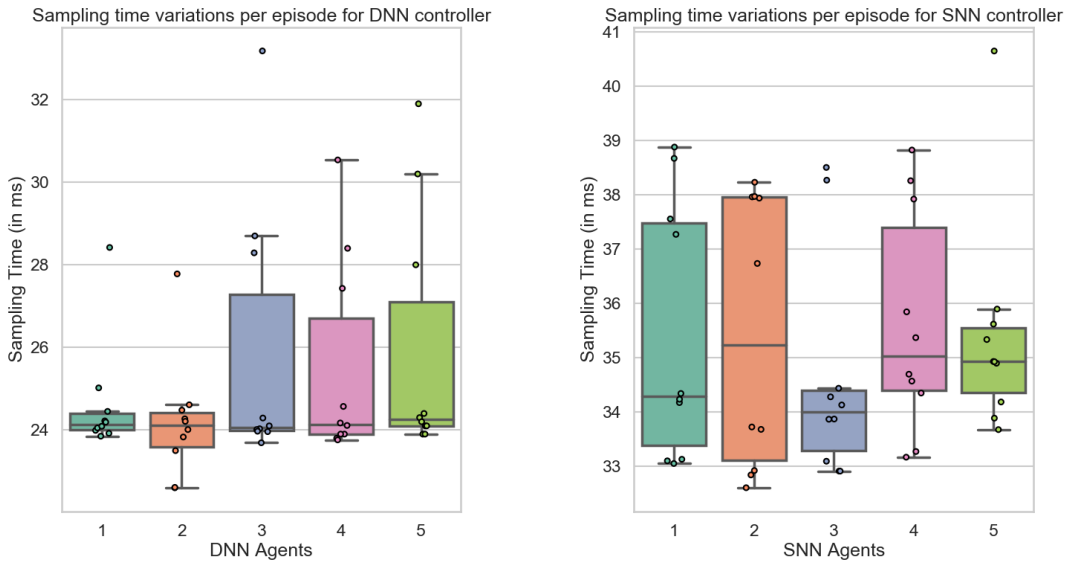
2. **State space drift:** This occurred mainly due to the construction mechanism of the cartpole system. A fairly simplistic mechanism coupled with inaccurate sensors and naive hardware resulted in producing a drift in the state space every time a new episode was initiated. We considered adding Gaussian noise into the state variables so that the AI controllers would become robust to the drifting state space.

3. **Skidding:** This was an issue that was encountered in the early stages of testing the hardware. The wheels as shown in A.2f of the LEGO Educational kit are made up of rubber that were skidding on a normal floor with consistent use over a period of time. To stop the wheel from skidding, we used a foamed polyethylene mat (commonly called a 'yoga' mat) that made the skidding almost non-existent. This allowed a direct translation of the controller's actions into physical movements with negligible delay.

4. **Slow Processor:** EV3 brick runs an ARM9 processor clocked at 300 Mhz. We used EV3 python on the Linux OS of the EV3 to develop the AI controllers. This setup combined

with the speed of the processor are not powerful enough to run large networks of size >50 neurons at a high speed e.g. <100 ms. This brought a major constraint on the size of the networks that we could use for our cartpole application. For a real cartpole system that we have built, it is required to provide actions in a few tens of milliseconds. If we keep the frequency of application of the actions higher, the control over the plant would become more robust and accurate. Therefore, after the initial experiments of using 40+ neurons (in DQN agent [4, 24, 16, 2]), we dropped the network size to a minimum of 10 neurons for obtaining a lower sampling time. However, there was a compromise in the quality of control with the decrease in the number of neurons.

We had come up with either the solutions to solve the problems or some workarounds to avoid the problem that occurred in the thesis due to the hardware. Despite that, the benchmark did not improve above an average balance time of 2.5 ms. The key reason that was hindering the performance of our controller was the inconsistent delays within the EV3. The cartpole problem is an inherently real-time problem in which we cannot take an arbitrary amount of time between two actions. Doing so would affect the performance and in worst cases can lead to controller failure as well. To prove the inconsistencies observed in the EV3 to execute the same configuration of the network, we show a box-plot describing the average sampling time for each of the controllers in figure 10.8.



(a) (b)

Figure 10.8: (a) Variations in sampling time for different DNN agents with the same network configuration, and (b) Variations in sampling time for different SNN agents with the same network configuration

Figure 10.8 visualizes the difference in the delays observed when running DNN and SNN controllers of the same network configurations on the hardware. In figure 10.8a, we have 5 different DNN agents with configuration [4, 4, 2] on the x-axis. For each DNN agent, we played 10 episodes and the average sampling time observed in each episode is plotted with a dot. Thus, we have a box plot for the average sampling time that each agent encountered, for 10 episodes. It is observed that the mean value of sampling time that every DNN agent had is around 25 ms. However, the variations of the sampling times are clearly evident. To exemplify, the DNN agent 3 had a sampling time of 33 ms in one of the episodes that it played on the hardware. The DNN agent 2 experienced an average sampling time of 22 ms as well as 28 ms in two different episodes on the

hardware. Since we measure the performance of an agent in terms of the average balance time for a few tens of episodes, the fact that one or more episodes were played at different sampling times cannot be ignored. It can happen that in those cases, due to the higher sampling time, the agent had a poor balancing time on the cartpole system.

Figure 10.8b describes the same details as mentioned above for the SNN agents. Each of the SNN agents had a configuration of [4, 4, 2] i.e. 10 neurons in its network. It can be observed that on an average the SNN controllers had an average sampling time of around 35 ms. However, the EV3 could not provide a sampling time of 35 ms consistently in every episode that an SNN agent played. For example, SNN agent 1 observed an average sampling time from 33 ms to 39 ms within 10 different episodes it played on the hardware.

The reasons for the inconsistencies observed in the EV3 can be attributed to non-real-time Linux OS that is uses. Some of the qualities that make Linux a non-real-time operating system include applications being considered as processes, I/Os being read via files, etc.

This justifies the performance of the controllers for having a notable lower benchmark of around 2.5 seconds of balance time. However, the fact that these agents were also able to balance the pole for a maximum of 6.03 seconds for a real-time task using non-real-time hardware makes our implementation in this thesis significant.

10.7 Conclusion

This research aimed at designing, implementing, optimizing, and evaluating energy-efficient AI controllers belonging to the category of Deep Neural Networks and Spiking Neural Networks. Further, the thesis also planned at testing the feasibility of the AI controllers on a practical real-time dynamical control system. As a result, we have designed and implemented a DNN and SNN controller using the Genetic Algorithm. Both the controllers executed a comparable performance on a real-life dynamical system that we built i.e a Cartpole. The practical cartpole was built from scratch and configured to be seen as a reinforcement learning problem which enabled the AI controllers to learn the task of balancing the pole by themselves. Based on the outcome of our implementations, we can say that the AI controllers are not only capable to perform exceptionally well on the simulated dynamical control systems but also execute control on a practical dynamical system with a real-time constraint. The only key requirement for the AI controllers to be able to execute control over a dynamical system is the real-time response of the hardware. In other words, the hardware must be equipped with a Real-Time Operating System (RTOS) in order to process and provide responses in a deterministic fashion. However, this requirement on the hardware is directly dependent on the nature of the dynamical system that has to be controlled. In our case, we had to control a real-time Cartpole system that requires a fixed and predetermined response time as the AI controllers encode this information into themselves while learning the task. With inconsistent responses from the hardware, the sampling time of the system gets affected directly and thereby degrading the controller performance. One of the key insights of our implementation was the effect of non-real-time hardware on the performance of the AI controllers. We observed that the AI controllers are affected drastically for a system with varying response times. Yet, the controllers that we designed exhibited their capabilities in controlling the real-time task of pole-balancing on a non-real-time hardware setup. We were also able to achieve the research goals set for the thesis as explained further.

A Deep Neural Network with a configuration of [4, 4, 2] (can also be termed as an ANN owing to a single hidden layered architecture) i.e. 4 input neurons, 4 neurons in the hidden layer, and 2 neurons at the output was capable of controlling a real Cartpole system with arguably significant performance. It set a benchmark for balancing the pole for an average of 2.5 seconds. We attribute the DNN controller to be robust as it could execute its control despite the system suffering from a

state-space drift. The mentioned controller used 30 32-bit float value parameters i.e. weight and biases to establish the control over the cartpole system by observing the data related to its cart movement (horizontal) and angular motion of the pole.

An SNN controller was also designed with a configuration exactly similar to the DNN controller i.e. [4, 4, 2]. It was able to present a comparable level of performance in controlling the same cartpole system as that of the DNN controller.

We built two energy models to profile and evaluate the DNN and SNN controllers. The first energy model was based on the energy used by the controller in terms of the hardware computations (processor core computation) and the memory transactions (Cache loads and stores). The second energy model was defined by the number of Multiply-Accumulate (MAC) operations that each of the controllers required. We conducted the energy profiling on a simulated system on the laptop instead of the real hardware due to the inflexible nature of the hardware controller (EV3) to support the energy measurements. The DNN controller was observed to be more energy-efficient than the SNN controller for the energy model based on internal energy consumptions required for computation. We observed the DNN controller to be 1.5 times energy-efficient than the SNN controller for controlling the same system and at the same level of performance. However, the SNN controller was observed to be 2.2 times computationally efficient according to the second energy model. The capability of SNN controllers to demonstrate sparse neuron firing helps them in performing fewer energy-expensive MAC operations. The DNNs on the other hand perform these MAC operations for each neuron in every feedforward process making them highly expensive in terms of computation.

To conclude, we present a diverse set of experiments to build and implement model-free, reinforcement learning algorithms, termed as AI controllers to efficiently control a real-world dynamical control system. We present a detailed energy comparison of the two categories of AI controllers namely DNN and SNN and discuss the results through this thesis.

10.8 Future scope and recommendations

Following are some of the recommendations that we provide based on our work for researchers interested to conduct a study on the topic relevant to our thesis:

- As a future study on the comparison of DNN and SNN controllers, we suggest using a Neuromorphic processor to realize the actual strength of Spiking Neural Networks over the DNNs. The platforms using neuromorphic processors are specially designed to execute neurobiological architectures such as SNNs. Keeping that in mind, we can compare and assess the energy consumptions on that platform for SNN and DNN controllers.

- We recommend using an RTOS over embedded Linux while designing real-time applications like a cartpole system on the hardware. Despite embedded Linux is popularly used in consumer embedded devices, its major functionality lies in performing non-real-time tasks used in touch screens, wireless routers, etc.

- We performed the energy comparison of the AI controllers by taking the ratio of the individual energy consumptions as this process was not carried out on the real cartpole system owing to hardware limitations. However, the scaling of these results to any hardware may bring slight variations in the results. As a future scope, we recommend using sophisticated real-world control plants that can allow direct measurements of energy consumptions on the system itself. This approach would help in producing an accurate comparison of energy consumptions of SNNs and DNNs in energy units rather than ratios.

Bibliography

[1] Seaborn boxplot. URL: https://seaborn.pydata.org/generated/seaborn.boxplot.html. 73

[2] tracemalloc - trace memory allocations. URL: https://docs.python.org/3/library/tracemalloc.html. 85

[3] Transparent, symmetric distributed computing. URL: https://rpyc.readthedocs.io/en/latest/. 25

[4] Working with ev3dev remotely using rpyc. URL: https://ev3dev-lang.readthedocs.io/projects/python-ev3dev/en/stable/rpyc.html. 25

[5] Understanding machine learning(as 6 jars):, Feb 2019. URL: https://mc.ai/understanding-machine-learningas-6-jars/. ix, 6

[6] M. Gethsiyal Augasta and T. Kathirvalavakumar. Pruning algorithms of neural networks — a comparative study. URL: https://www.researchgate.net/publication/257471847_Pruning_algorithms_of_neural_networks_-_a_comparative_study. 70

[7] Martin P Dimitrov. Intel® power governor. URL: https://software.intel.com/content/www/us/en/develop/articles/intel-power-governor.html. x, 88

[8] LEGO Education. Mindstorms ev3 developer kits – support – lego education. URL: https://education.lego.com/en-us/support/mindstorms-ev3/developer-kits. xii, xii, xii, 22, 23

[9] Emad Elbeltagi, Tarek Hegazy, and Donald Grierson. Comparison among five evolutionary-based optimization algorithms, Jul 2005. URL: https://www.sciencedirect.com/science/article/pii/S1474034605000091. 15

[10] Wulfram Gerstner, Werner M. Kistler, Richard Naud, and Liam Paninski. *Neuronal dynamics from single neurons to networks and models of cognition.* Cambridge University Press, 2016. 19

[11] D. Hackenberg, T. Ilsche, R. Schone, D. Molka, M. Schmidt, and W. E. Nagel. Power measurement techniques on standard compute nodes: A quantitative comparison. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS 2013)*, pages 194–204, Los Alamitos, CA, USA, apr 2013. IEEE Computer Society. URL: https://doi.ieeecomputersociety.org/10.1109/ISPASS.2013.6557170, doi:10.1109/ISPASS.2013.6557170. 87

[12] Song Han, Jeff Pool, John Tran, and William J. Dally. Learning both weights and connections for efficient neural networks, Oct 2015. URL: https://arxiv.org/abs/1506.02626. x, 70

[13] A L HODGKIN and A F HUXLEY. A quantitative description of membrane current and its application to conduction and excitation in nerve, Aug 1952. URL: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC1392413/?page=1. 18

[14] John H. Holland, Luis M. Rocha, Jordan Pollack, Mark A. Bedau, Phil Husbands, Richard A. Watson, and Takashi Ikegami. Adaptation in natural and artificial systems. URL: https://mitpress.mit.edu/books/adaptation-natural-and-artificial-systems. 15

[15] Doug Laney. Application delivery strategies - gartner blog network. URL: https://blogs.gartner.com/doug-laney/files/2012/01/ad949-3D-Data-Management-Controlling-Data-Volume-Velocity-and-Variety.pdf. 5

[16] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning, Dec 2013. URL: https://arxiv.org/abs/1312.5602. 10, 11, 29

[17] Priyadarshini Panda, Aparna Aketi, and Kaushik Roy. Towards scalable, efficient and accurate deep spiking neural networks with backward residual connections, stochastic softmax and hybridization, Oct 2019. URL: https://arxiv.org/abs/1910.13931. 93

[18] M Riedmiller. Concepts and facilities of a neural reinforcement learning control architecture for technical process control, Feb 2014. URL: https://link.springer.com/article/10.1007/s005210050038. 12

[19] M. Riedmiller and H. Braun. A direct adaptive method for faster backpropagation learning: the rprop algorithm. In *IEEE International Conference on Neural Networks*, pages 586–591 vol.1, 1993. 12

[20] Martin Riedmiller. 10 steps and some tricks to set up neural reinforcement ... URL: http://ml.informatik.uni-freiburg.de/former/_media/publications/12riedmillertricks.pdf. 13, 48, 49, 54, 55, 95

[21] Martin Riedmiller. Neural reinforcement learning to swing-up and balance a ... URL: http://ml.informatik.uni-freiburg.de/_media/publications/riesmc05.pdf. 12, 13, 14, 43, 54

[22] M.G. Rodd and G.J. Suski. *Artificial Intelligence in Real-Time Control 1991: Proceedings of the 3rd IFAC Workshop, California, USA, 23-25 September 1991*. ISSN. Elsevier Science, 2014. URL: https://books.google.nl/books?id=F9LSBQAAQBAJ. 29

[23] Tim Salimans, Jonathan Ho, Xi Chen, Szymon Sidor, and Ilya Sutskever. Evolution strategies as a scalable alternative to reinforcement learning, 2017. arXiv:1703.03864. 62

[24] Tom Schaul, Justin Bayer, Daan Wierstra, Yi Sun, Martin Felder, Frank Sehnke, Thomas Rückstieß, and Jürgen Schmidhuber. PyBrain. *Journal of Machine Learning Research*, 2010. 27

[25] Richard S. Sutton and Andrew G. Barto. *Reinforcement learning: an introduction*. The MIT Press, 2018. ix, 7

[26] Keras Team. Keras documentation. URL: https://keras.io/why_keras/. 26

[27] Jannick Verlie. Control of an inverted pendulum with deep reinforcement learning. URL: https://lib.ugent.be/fulltxt/RUG01/002/367/558/RUG01-002367558_2017_0001_AC.pdf. 41

[28] Bojian Yin, Federico Corradi, and Sander M. Bohté. Effective and efficient computation with multiple-timescale spiking recurrent neural networks, Jun 2020. URL: https://arxiv.org/abs/2005.11633. ix, 19, 93

Appendix A

LEGO MINDSTORMS EV3 component images

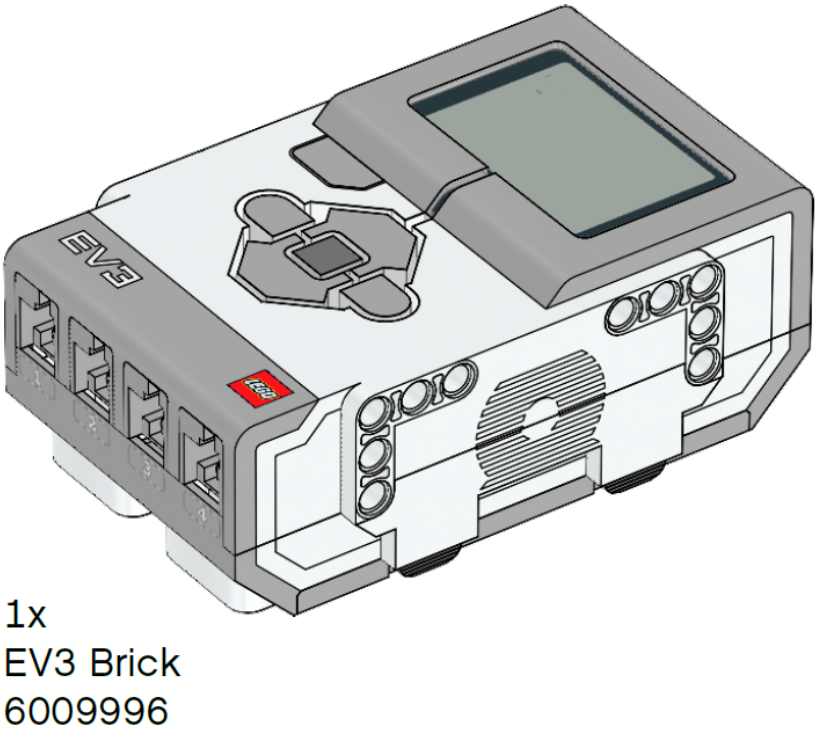
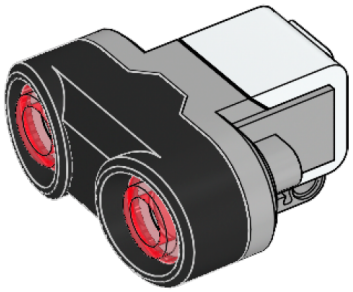
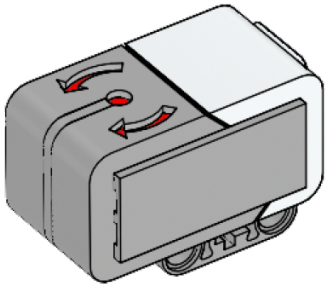


Figure A.1: EV3 Brick



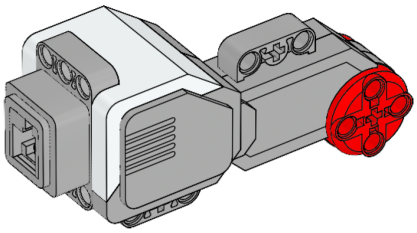
1x
Ultrasonic Sensor
6008924

(a) EV3 Ultrasonic Sensor



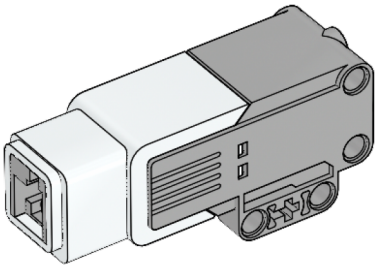
1x
Gyro Sensor
6008916

(b) EV3 Gyro Sensor



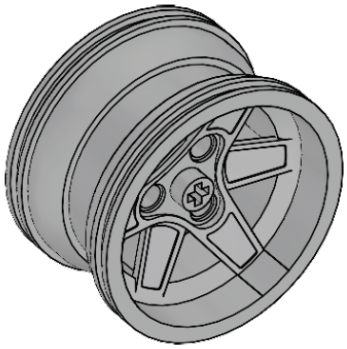
2x
Large Motor
6009430

(c) EV3 Large Motor



1x
Medium Motor
6008577

(d) EV3 Medium Motor



4x
4211845

(e) Wheel rim

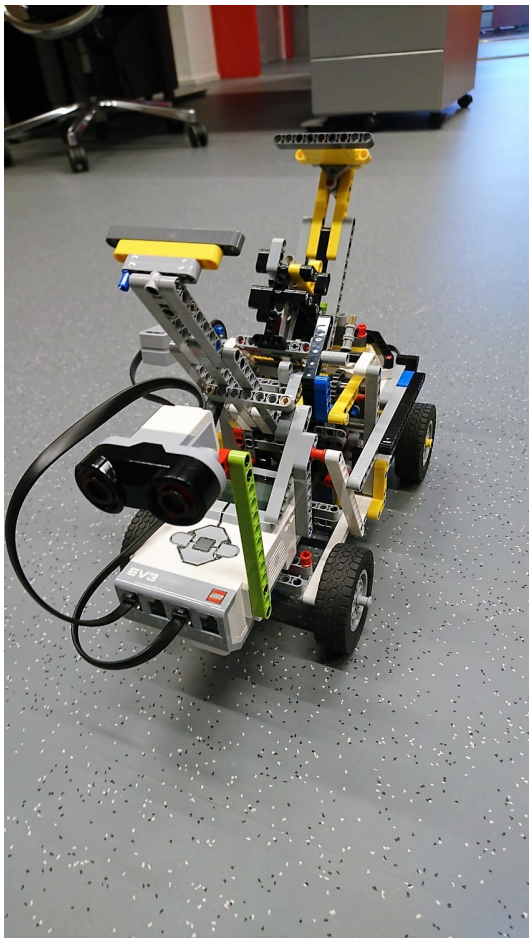
4x
4614801

(f) Wheel tyres

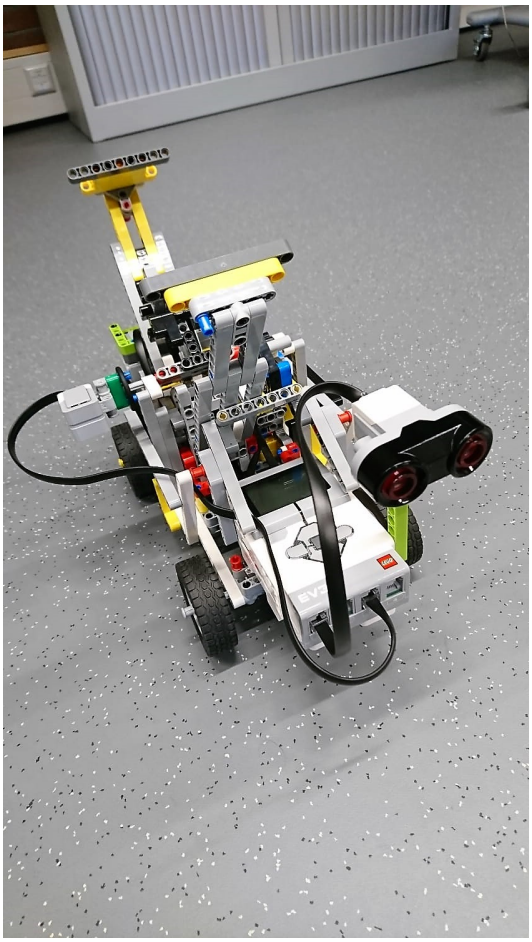
Figure A.2: Hardware components of the Cartpole System

Appendix B

Cartpole System Images

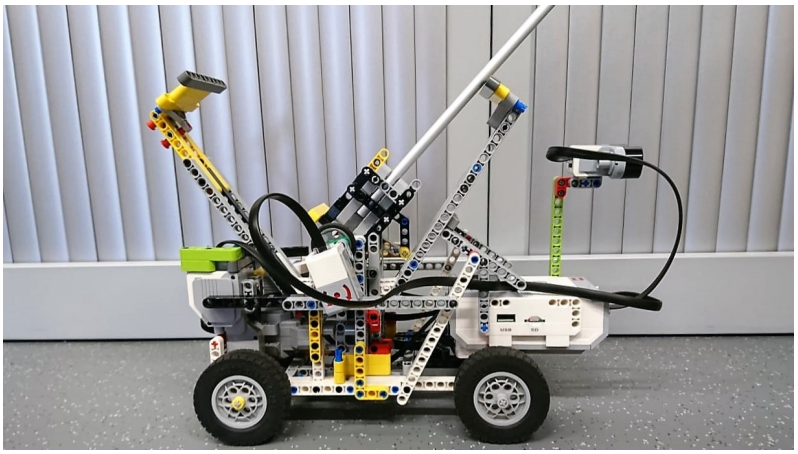


(a) Two-point perspective(1)

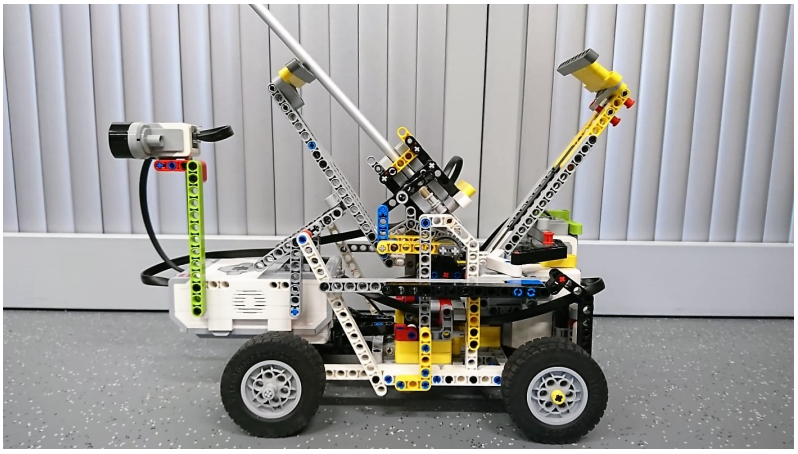


(b) Two-point perspective(2)

Figure B.1: Cart realization

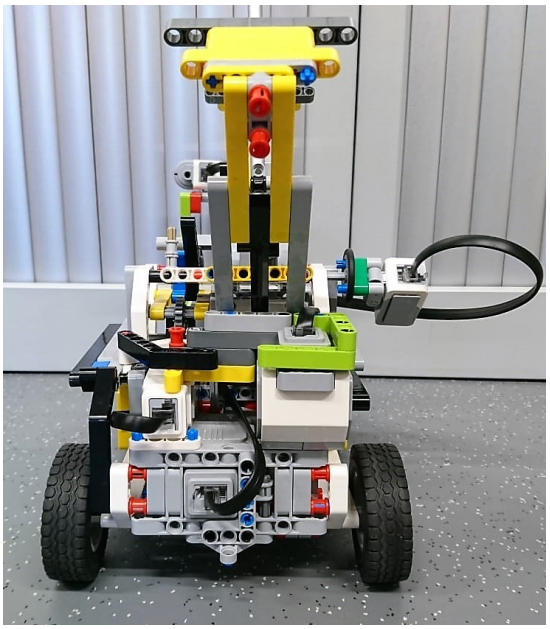


(a) Right side view



(b) Left side view

(c) Front view



(d) Rear view

Figure B.2: Pictorial views of the Cartpole System

Appendix C

T-Table

cum. prob	$t_{.50}$	$t_{.75}$	$t_{.80}$	$t_{.85}$	$t_{.90}$	$t_{.95}$	$t_{.975}$	$t_{.99}$	$t_{.995}$	$t_{.999}$	$t_{.9995}$
one-tail	0.50	0.25	0.20	0.15	0.10	0.05	0.025	0.01	0.005	0.001	0.0005
two-tails	1.00	0.50	0.40	0.30	0.20	0.10	0.05	0.02	0.01	0.002	0.001
df											
1	0.000	1.000	1.376	1.963	3.078	6.314	12.71	31.82	63.66	318.31	636.62
2	0.000	0.816	1.061	1.386	1.886	2.920	4.303	6.965	9.925	22.327	31.599
3	0.000	0.765	0.978	1.250	1.638	2.353	3.182	4.541	5.841	10.215	12.924
4	0.000	0.741	0.941	1.190	1.533	2.132	2.776	3.747	4.604	7.173	8.610
5	0.000	0.727	0.920	1.156	1.476	2.015	2.571	3.365	4.032	5.893	6.869
6	0.000	0.718	0.906	1.134	1.440	1.943	2.447	3.143	3.707	5.208	5.959
7	0.000	0.711	0.896	1.119	1.415	1.895	2.365	2.998	3.499	4.785	5.408
8	0.000	0.706	0.889	1.108	1.397	1.860	2.306	2.896	3.355	4.501	5.041
9	0.000	0.703	0.883	1.100	1.383	1.833	2.262	2.821	3.250	4.297	4.781
10	0.000	0.700	0.879	1.093	1.372	1.812	2.228	2.764	3.169	4.144	4.587
11	0.000	0.697	0.876	1.088	1.363	1.796	2.201	2.718	3.106	4.025	4.437
12	0.000	0.695	0.873	1.083	1.356	1.782	2.179	2.681	3.055	3.930	4.318
13	0.000	0.694	0.870	1.079	1.350	1.771	2.160	2.650	3.012	3.852	4.221
14	0.000	0.692	0.868	1.076	1.345	1.761	2.145	2.624	2.977	3.787	4.140
15	0.000	0.691	0.866	1.074	1.341	1.753	2.131	2.602	2.947	3.733	4.073
16	0.000	0.690	0.865	1.071	1.337	1.746	2.120	2.583	2.921	3.686	4.015
17	0.000	0.689	0.863	1.069	1.333	1.740	2.110	2.567	2.898	3.646	3.965
18	0.000	0.688	0.862	1.067	1.330	1.734	2.101	2.552	2.878	3.610	3.922
19	0.000	0.688	0.861	1.066	1.328	1.729	2.093	2.539	2.861	3.579	3.883
20	0.000	0.687	0.860	1.064	1.325	1.725	2.086	2.528	2.845	3.552	3.850
21	0.000	0.686	0.859	1.063	1.323	1.721	2.080	2.518	2.831	3.527	3.819
22	0.000	0.686	0.858	1.061	1.321	1.717	2.074	2.508	2.819	3.505	3.792
23	0.000	0.685	0.858	1.060	1.319	1.714	2.069	2.500	2.807	3.485	3.768
24	0.000	0.685	0.857	1.059	1.318	1.711	2.064	2.492	2.797	3.467	3.745
25	0.000	0.684	0.856	1.058	1.316	1.708	2.060	2.485	2.787	3.450	3.725
26	0.000	0.684	0.856	1.058	1.315	1.706	2.056	2.479	2.779	3.435	3.707
27	0.000	0.684	0.855	1.057	1.314	1.703	2.052	2.473	2.771	3.421	3.690
28	0.000	0.683	0.855	1.056	1.313	1.701	2.048	2.467	2.763	3.408	3.674
29	0.000	0.683	0.854	1.055	1.311	1.699	2.045	2.462	2.756	3.396	3.659
30	0.000	0.683	0.854	1.055	1.310	1.697	2.042	2.457	2.750	3.385	3.646
40	0.000	0.681	0.851	1.050	1.303	1.684	2.021	2.423	2.704	3.307	3.551
60	0.000	0.679	0.848	1.045	1.296	1.671	2.000	2.390	2.660	3.232	3.460
80	0.000	0.678	0.846	1.043	1.292	1.664	1.990	2.374	2.639	3.195	3.416
100	0.000	0.677	0.845	1.042	1.290	1.660	1.984	2.364	2.626	3.174	3.390
1000	0.000	0.675	0.842	1.037	1.282	1.646	1.962	2.330	2.581	3.098	3.300
z	0.000	0.674	0.842	1.036	1.282	1.645	1.960	2.326	2.576	3.090	3.291
	0%	50%	60%	70%	80%	90%	95%	98%	99%	99.8%	99.9%
	Confidence Level										

Figure C.1: The T-table.