REINFORCEMENT LEARNING FOR POWER SCHEDULING IN A GRID-TIED PV-BATTERY ELECTRIC VEHICLES CHARGING STATION

By

Erick Odhiambo Arwa

Department of Electrical Engineering

University of Cape Town

A dissertation presented for the award of the degree of

Master of Science in Electrical Engineering.

Supervisor: Prof. Komla A. Folly

October 2020

University of Cape Town

University of Cape Town

The copyright of this thesis vests in the author. No quotation from it or information derived from it is to be published without full acknowledgement of the source. The thesis is to be used for private study or non-commercial research purposes only.

Published by the University of Cape Town (UCT) in terms of the non-exclusive license granted to UCT by the author.

Declaration

I know the meaning of plagiarism and declare that all the work in the document, save for that which is properly acknowledged, is my own. This dissertation has been submitted to the Turnitin module and I confirm that my supervisor has seen my report and any concerns revealed by such have been resolved with my supervisor.

Signed by candidate

Sign: …………………………

Erick Odhiambo Arwa

October 30, 2020

Dedication

To my father, Mr. Joseph Arwa Opuch (Japuonj) and my grandmother, Mrs. Sophia Saoke (Nyodiala).

Acknowledgment

I am deeply grateful to God for guiding me through this work. Unto Him be all glory and honor, now and forever.

My deep appreciation goes to my supervisor, Prof. Komla A. Folly for his loving and unyielding support throughout my journey in this study.

I thank Mandela Rhodes Foundation for fully sponsoring my studies. I am grateful for the leadership, entrepreneurship and reconciliation workshops that challenged me to take my interaction with people to a meaningful level.

I also thank my friends Nicodemus Abungu, Philemon Arito, Beryl Arito, Innocent Safari, Dorcas Safari, Grace Awuor and Athenkosi Nzala whose love, tenderness and care brought a family experience to my studies.

Many thanks to uncles Joshua Saoke, Stephen Saoke, Paul Saoke and Dickson Otora for their invaluable contribution to my education journey, particularly during this study. I am grateful to Auntie Jacky Saoke too for her kind support and encouragement.

Finally, I am grateful to the University of Cape Town's Power Systems research group for their candid feedback on my presentations regarding the concepts developed in this dissertation.

Abstract

Grid-tied renewable energy sources (RES) based electric vehicle (EV) charging stations are an example of a distributed generator behind the meter system (DGBMS) which characterizes most modern power infrastructure. To perform power scheduling in such a DGBMS, stochastic variables such as load profile of the charging station, output profile of the RES and tariff profile of the utility must be considered at every decision step. The stochasticity in this kind of optimization environment makes power scheduling a challenging task that deserves substantial research attention.

This dissertation investigates the application of reinforcement learning (RL) techniques in solving the power scheduling problem in a grid-tied PV-powered EV charging station with the incorporation of a battery energy storage system. RL is a reward-motivated optimization technique that was derived from the way animals learn to optimize their behavior in a new environment. Unlike other optimization methods such as numerical and soft computing techniques, RL does not require an accurate model of the optimization environment in order to arrive at an optimal solution. This study developed and evaluated the feasibility of two RL algorithms, namely, an asynchronous Q-learning algorithm and an advantage actor-critic (A2C) algorithm, in performing power scheduling in the EV charging station under static conditions. To assess the performances of the proposed algorithms, the conventional Q-learning and actor-critic algorithm were implemented to compare their global cost convergence and their learning characteristics.

First, the power scheduling problem was expressed as a sequential decision-making process. Then an asynchronous Q-learning algorithm was developed to solve it. Further, an advantage actor-critic (A2C) algorithm was developed and was used to solve the power scheduling problem. The two algorithms were tested using a 24-hour load, generation and utility grid tariff profiles under static optimization conditions. The performance of the asynchronous Q-learning algorithm was compared with that of the conventional Q-learning method in terms of the global cost, stability and scalability. Likewise, the A2C was compared with the conventional actor-critic method in terms of stability, scalability and convergence. Simulation results showed that both the developed algorithms (asynchronous Q-learning algorithm and A2C) converged to lower global costs and displayed more stable learning characteristics than their conventional counterparts.

This research established that proper restriction of the action-space of a Q-learning algorithm improves its stability and convergence. It was also observed that such a restriction may come with compromise of computational speed and scalability. Of the four algorithms analyzed, the A2C was found to produce a power schedule with the lowest global cost and the best usage of the battery energy storage system.

Table of Contents

Declaration i

Dedication ii

Acknowledgment iii

Abstract iv

Figures xi

Tables xiii

Abbreviations and Notations xiv

1.0 Introduction 1

1.1 Research Background 1

1.1.1 Electric Vehicle Industry 1

1.1.2 Environmental Concerns in the EV Adoption 2

1.1.3 Power Scheduling in Grid-Tied EV Charging Stations 2

1.2 Problem statement 3

1.3 Reinforcement Learning 4

1.4 Research Motivation 5

1.5 Research Objectives 6

1.6 Expected Outcomes of the Research 6

1.7 Limitations and Scope 6

1.8 Dissertation outline 7

1.9 Research Outputs 8

2.0 A Review of Power Scheduling Techniques in Grid-Connected Microgrids 10

2.1 Background 10

2.2 Power Scheduling Problem Definition 12

2.2.1 Setup of a Grid-Connected PV/battery EV Charging Station 12
2.2.2 Mathematical Formulations of the Power Management Problem 13
2.3 Priority Listing 14
2.4 Linear Search Methods 15
2.5 Global Search Methods 16
2.5.1 Evolutionary Algorithms 16
2.5.2 Swarm Intelligence 17
2.6 Reinforcement Learning Techniques 18
2.6.1 Dynamic Programming 20
2.6.2 Q-learning 20
2.6.3 Batch Reinforcement Learning 21
2.6.4 Deep Q-network 22
2.6.5 Policy Gradients Methods 23
2.7 Summary 25
3.0 Reinforcement Learning: A Reward-Based Control Process 27
3.1 Background 27
3.2 Markov Decision Process and Components of Reinforcement Learning 29
3.2.1 State and State Space 30
3.2.2 Action and Action Space 31
3.2.3 State Transition 31
3.2.4 Reward/Reinforcement Function 31
3.2.5 Policy 32
3.2.6 Value function 32
3.3 Solving Markov Decision Processes 34
3.3.1 Value Iteration 35
3.3.2 Policy Iteration 35
3.4 Reinforcement Learning Techniques 35
3.4.1 Monte Carlo Learning 35
3.4.2 Temporal Difference Learning 36
3.4.3 Q-learning 36

3.4.4 Action Selection Strategies ... 38
3.4.5 Fitted Q-iteration: Q-learning with Function Approximation ... 39
3.4.6 Batch Reinforcement Learning ... 43
3.4.7 Deep Q-network ... 43
3.4.8 Policy Gradient Theorem ... 44
3.4.9 Actor-Critic Architectures ... 46
3.5 Summary ... 47

4.0 Q-Learning for Power Scheduling in Grid-Tied EV Charging Station ... 49

4.1 Background ... 49
4.2 Mathematical Formulation of the Power Scheduling Problem ... 51
4.2.1 Charging Station Model ... 51
4.2.2 Objective Function ... 52
4.3 Grid-tied Microgrid Power Scheduling as a Markov Decision Process (MDP) ... 58
4.4 Conventional Q-learning Solution to the Power Scheduling Problem ... 62
4.5 Proposed Asynchronous Q-learning Solution for the Power Scheduling Problem ... 64
4.6 Summary ... 66

5.0 Power Scheduling Using Advantage Actor-Critic Algorithm ... 67

5.1 Background ... 67
5.2 Conventional Actor-critic Algorithm ... 68
5.3 Advantage Actor-critic Algorithm ... 69
5.4 Power Scheduling Using Advantage Actor-Critic Algorithm ... 72
5.4.1 Environment design ... 72
5.4.2 Model of the Actor and Critic Networks ... 73
5.4.3 Training of A2C ... 74
5.5 Summary ... 75

6.0 Simulation Results and Analysis ... 76

6.1 Simulation Platform ... 76
6.2 Input Data ... 76

6.2.1 Charging system parameters ... 76
6.2.2 Test Data ... 77
6.3 Simulation Results for Asynchronous Q-learning ... 78
6.3.1 Learning Parameters ... 78
6.3.2 Learning Characteristics of Q-learning ... 79
6.3.3 Effect of the Reward Signal on Q-learning ... 82
6.3.4 Global Cost Convergence of Q-learning ... 84
6.3.5 Optimized Power Schedule: Asynchronous Q-learning ... 88
6.3.6 Scalability of the Q-learning Algorithms ... 90
6.4 Simulation Results of A2C Algorithm ... 93
6.4.1 Learning Parameters ... 93
6.4.2 Learning Characteristics and Cost Convergence ... 94
6.4.3 Impact of reward function on A2C and AC ... 97
6.4.4 Optimized Power Schedule: A2C ... 98
6.4.5 Scalability Analysis for A2C ... 101
6.5 Discussions ... 104
6.5.1 Discussions on the Power Schedule Results ... 104
6.5.2 Algorithmic Limitations ... 105
6.6 Summary ... 106
7.0 Conclusions and Recommendations ... 108
7.1 Conclusions ... 108
7.2 Recommendations ... 109
References ... 111
Appendices ... 123
Appendix A: The Policy Gradient Theorem and the A2C Algorithm ... 123
A.1: Proof of Policy Gradient Theorem ... 123
A.2: Impact of Number of Nodes in the Hidden Layer ... 125
Appendix B: Load, PV, and Grid Tariff Data ... 126
Appendix C: Simulation Tools ... 128

Appendix D: Python Codes 128
D.1: Learning Environment Code 128
D.2: Conventional Q-learning Code 130
D.3: Asynchronous Q-learning Code 132
D.4: Conventional Actor-critic Code 135
D.5: Advantage Actor-Critic Code 138

Figures

Fig. 2.1: Bus system configurations for an EV charging station: (a) DC bus system (b) AC bus system 13
Fig. 2.2: Data flow diagram for an MPC algorithm........ 16
Fig. 2.3: An illustration of an agent-environment interaction for optimal power scheduling 19
Fig. 3.1: An illustration of an RL agent-environment interaction 28
Fig. 3.2: High level illustration of an RL Environment........ 30
Fig. 3.3:Q-learning algorithm 38
Fig. 3.4: Anatomy of a single neuron in an artificial neural network 41
Fig. 3.5: Data flow diagram for a batch reinforcement learning algorithm 43
Fig. 3.6: Data flow diagram for an actor-critic algorithm........ 47
Fig. 4.1:Model of the EV charging station 51
Fig. 4.2: Action space determination algorithm........ 60
Fig. 4.3: Conventional Q-learning for power scheduling 63
Fig. 4.4: Asynchronous Q-learning for power scheduling........ 65
Fig. 4.5: Policy retrieval algorithm 66
Fig. 5.1: Conventional Actor-Critic Algorithm 70
Fig. 5.2: Advantage actor-critic algorithm........ 71
Fig. 5.3: A2C actor model........ 74
Fig. 5.4: A2C critic model 74
Fig. 6.1: Charging station's load, generation profile and the grid tariff profile 78
Fig. 6.2: Learning characteristics for the conventional Q-learning 80
Fig. 6.3: Learning characteristics for the proposed asynchronous Q-learning 81
Fig. 6.4: Learning curves for the asynchronous Q-learning and the conventional Q-learning..... 82
Fig. 6.5: Episodic moving average cost profiles for the conventional Q-learning under different reward functions........ 84
Fig. 6.6: Episodic moving average cost profiles for the asynchronous Q-learning under different reward functions........ 84
Fig. 6.7: Episodic cost profile for the conventional Q-learning method 85
Fig. 6.8: Episodic cost profile for the proposed asynchronous Q-learning method 85
Fig. 6.9: Comparative cost profile for the asynchronous and the conventional Q-learning......... 86
Fig. 6.10: Optimized battery energy profiles for both Q-learning algorithms........ 88

Fig. 6.11: Optimized power schedule obtained using the asynchronous Q-learning algorithm (battery energy values in kWh divided by time to get kW values)... 89
Fig. 6.12: A plot of computational time against battery size for the conventional Q-learning 92
Fig. 6.13: A plot of computational time against battery size for the asynchronous Q-learning ... 93
Fig. 6.14: Learning curve for the conventional AC algorithm 96
Fig. 6.15: Learning curve for the A2C algorithm 96
Figure 6.16: Episodic reward profiles for the conventional AC and the A2C algorithms............ 97
Figure 6.17: Episodic cost profile for conventional AC algorithm under different reward functions 98
Fig. 6.18: Episodic cost profile for A2C algorithm under different reward functions 98
Fig. 6.19: Optimized power schedule obtained from the A2C algorithm ((battery energy values in kWh divided by time to get kW values) 100
Fig. 6.20: Optimized battery energy profiles for both the A2C and the AC algorithms 101
Fig. 6.21: A plot of computational time against battery size for the conventional actor-critic algorithm 103
Fig. 6.22: A plot of computational time against battery size for the A2C algorithm 103
Fig. A: Learning curves for A2C using different number of nodes on the hidden layer of the actor and the critic models 126

Tables

Table 4.1: Q-table for the conventional Q-learning method 62
Table 6.1: Charging station system parameters 77
Table 6.2: Learning parameters 79
Table 6.3: Returned costs for both algorithms 87
Table 6.4: A2C learning parameters 94
Table 6.5: Global costs, battery degradation cost and the STD of battery energy profile for A2C and AC 100
Table 6.6: Summary of the Power Schedule Costs, Battery Energy Usage and Scalability of the Algorithms 105
Table B: Input Data 126
Table C: Python tools for machine learning 128

Abbreviations and Notations

Abbreviations

AC	Actor-critic
A2C	Advantage actor-critic
A3C	Asynchronous advantage actor-critic
AI	Artificial intelligence
ANN	Artificial neural network
BSS	Battery storage system
BRL	Batch reinforcement learning
CPP	Critical peak pricing
CS	Charging station
DDPG	Deep deterministic policy gradient
DGBMS	Distributed generator behind meter system
DNN	Deep neural network
DoD	Depth of discharge
DP	Dynamic programming
DPG	Deep policy gradient
DQN	Deep Q-network
DSO	Distribution system operator
ESNN	Eco-state neural network
ESS	Energy storage system
EV	Electric vehicle
EVSE	Electric vehicle supply equipment
GA	Genetic algorithm
GHG	Greenhouse gas
GT-MG	Grid-tied microgrid
HVAC	Heating, ventilation and air-conditioning
ICEV	Internal combustion engine vehicle
MDP	Markov decision process
MG	Microgrid
MILP	Mixed integer linear programming
MPC	Model predictive control

MPPT	Maximum power point tracking
OPF	Optimal power flow
PO-MDP	Partially observable Markov decision process
PSO	Particle swarm optimization
PV	Photovoltaic
ReLU	Rectified linear unit
RES	Renewable energy source
RL	Reinforcement learning
RTP	Real-time pricing
SoC	State of charge
SoH	State of health
STD	Standard deviation
TD	Temporal difference
ToU	Time of use
VDBE	Value difference-based exploration

Notations

a	Action
A	Advantage function
$\mathcal{A}$	Action space
C_{bd}	Battery degradation cost
C_{bt}	Battery capital cost per kWh
C_{DoD}	Battery degradation cost due to depth of discharge
$C_{P_{bss}}$	Battery power cost
C_{P_g}	Grid power cost
C_T	Cost of battery degradation due to temperature
DoD	Depth of discharge
∂	Temporal difference error
E_b	Battery energy at full charge
$\mathbb{E}$	Expectation operator
G_t	Grid tariff
J	Agent’s learning objective

j	Depth of discharge index
k	Timestep/state index
L	Loss function
$L_t(T)$	Battery lifetime as a function of ambient temperature
$L(DoD)$	Battery lifetime as a function of depth of discharge
n	Iteration index
P	Probability distribution
P_{cl}	Load at the charging station
P_g	Grid power
P_{pv}	PV power
P_{bss}	Battery charge/discharge power
$\mathcal{P}$	Neural network agent policy variable
Q	Q-value
Q_{fade}	Battery capacity fade
$\mathcal{R}$	Total rewards for a specific number of steps or episode
r	Reward
R_t	Battery thermal resistance
SoC	Battery state of charge
SoC_{av}	Average state of charge
SoE	State of energy (battery energy level)
T	Battery pack temperature
T_a	Ambient temperature
t	Time
x	State variable
α	Critic learning rate (also learning rate in Q-learning)
β	Actor learning rate
γ	Discount factor
θ	Neural network parameter vector (used for critic neural network)
π	Policy function
V	Value function
ω	Neural network parameter vector (use for actor neural network)
χ	State space

Y_h	Number of hours in a year
$\mathbb{Z}$	A sequence of transitions

1.0 Introduction

This Chapter presents an introduction to the problem under investigation. The Chapter provides a brief background to the electric vehicle (EV) industry, the environmental concerns in the adoption of electric vehicles and the need for proper power management in the EV charging stations. Key aspects of the research such as the problem statement, the research objectives and the research limitations have also been presented.

1.1 Research Background

1.1.1 Electric Vehicle Industry

An electric vehicle (EV) is an automobile that uses one or more electric motors to drive the wheels using stored electrical energy. It consists of a motor, a controller (with multiple power converters), a power source/storage, and a transmission system [1]. EVs were introduced into the transport industry in the early 19th century but faced a myriad of challenges in the market. Some of the challenges included poor battery performance, short range (distance covered on one charge), low maximum speed and inability to surmount steep gradients [2].

However, in the first quarter of the 20th century, significant advances in battery technology led to an improvement in the storage capacity of EV battery packs by at least 300%. This led to an improvement of the EV range by 230%. These new developments made the EVs to gain a substantial market share [3]. In addition, there are concerns that the deposits of fossil fuels are getting exhausted. Also, there is a considerable increase in the emission of green-house gases due to the use of internal combustion engine vehicles (ICEVs). Thus, if they were supplied from renewable energy sources, EVs would be a better alternative to ICEVs [4].

Apart from their environmental benefits, modern EVs come with technical and economic advantages. The technical advantages of EVs over ICEVs include the ease of speed control, a wider speed range, uniformity of power and torque output and a high power density. Economically, EVs are cheaper to maintain since electricity energy prices are lower than the fossil fuel equivalent [4]. Despite the marked merits of EV adoption, there are still outstanding concerns in the EV industry. The concerns include the range anxiety, the high-power requirements for fast EV charging. Furthermore, EV charging would potentially lower the quality of supply of power from the utility grids [5].

1.1.2 **Environmental Concerns in the EV Adoption**

It is estimated that the transportation sector causes at least 19% of CO_2 emissions [3]. Currently, the transport sector accounts for up to 53% of fossil fuel consumption worldwide [6]. The adoption of EVs for vehicular transport is seen as a means of reducing the overdependence on fossil fuels and to lower the GHG emissions.

Nonetheless, research has shown that there is a considerable increase in GHG emission associated with EV if their adoption is not organically infused with renewable energy development. Liu *et al.* [7] studied the possible effects of replacing the ICEVs with EVs and charging the EVs using power from the utility grid in South Africa. Using life cycle emissions analysis, the authors established that the replacement of ICEVs with EVs would lead to about 90% more GHG emissions. Life cycle (well-to-wheel) emissions analysis is done by estimating the amount of emission that may result from the energy transformation starting from the fossil fuel source (well) to the wheel of an EV. Moreover, the overall life cycle efficiency of converting fossil fuel at a power station to the wheel energy for the EVs is only just about 5% higher than for ICEVs [1]. Thus, there is a need to incorporate renewable energy sources (RES) in the charging of EVs.

1.1.3 **Power Scheduling in Grid-Tied EV Charging Stations**

There is a global increase in the number of EV charging stations being connected to the utility grids. The current electric grids have structural limitations such as insufficient generation infrastructure, limited transmission line capacity, etc. If this ramped up connection is sustained, there will be a necessity to upgrade the utilities to meet the rapidly increasing load demand. This upgrade of the utility infrastructure may be exceedingly expensive. Also, the upgrade may be too slow to catch up with the surge in the load demand.

RES-powered microgrids integrated with energy storage systems (ESS), have been reported to offer cheap, low emission and steady charging system for EVs. The deployment of off-grid microgrid systems would also alleviate the expected power quality constraints on the grid [8]. Considering the intermittency of RES generation, it is necessary to connect the microgrids to the main grid as a backup. Although energy storage systems may help address the intermittency, the current viable storage technologies are still expensive. The connection of EV charging stations to the utility grid has been reported to produce significant harmonic distortions, voltage instability and as well as an increase in the load demand that may cause local strains on the power network

[7], [9]. Furthermore, power dispatch and control in grid-connected microgrids is a challenging task due to the intermittency of load and some generation systems on both the microgrid and the utility grid sides [10].

Recently, studies on distributed generations have shown that grid-tied microgrids, if properly controlled, may help to improve the resilience, power quality and overall reliability of the utility grid [10]. There is, therefore, a need to employ robust power flow control mechanisms to ensure smooth energy cooperation between the RES-powered charging stations and the utility grid. The objectives of a power scheduling algorithm are to minimize the cost of energy production and delivery, to minimize power losses, to curtail load shedding, to maximize the overall system performance, etc.

1.2 Problem statement

Optimal power scheduling involves the temporal arrangement of the system's power resources to achieve the system's objective and maintain its overall health [11]. Conventional power systems scheduling tasks include economic dispatch, unit commitment and automatic generation control [11]. For the case of a RES-based, grid-tied EV charging station, the optimal scheduling algorithm is supposed to decide when and how much power is to be drawn from each of the station's power resources in order to supply the station's load at minimum cost and within the system's constraints. The cost of charging an EV that is incurred by the owner of the station is the sum of the cost of power purchase from the grid and the cost of degradation of the battery storage system (BSS) [12].

The design of an optimal power scheduling algorithm for a grid-tied RES-powered EV charging station boils down to the following questions:

i) How can the charging station be operated in order to meet the load demand while minimizing the operational cost?
ii) How can the algorithm schedule the power resources so that grid power is purchased when it is cheap, and the battery is operated at power levels that do not significantly reduce its cycle life?

In this study, a grid-tied EV charging with a battery storage system (BSS) is considered for power scheduling over a 24-hour optimization horizon. In each hour within this time frame, the scheduling algorithm is to determine the magnitude and direction of the power flow for each of

the power resources of the charging station. The sources of power for the charging station include the solar PV generator, the utility grid and the BSS. The BSS and the utility are able to supply the demand at the charging station as well as absorb any extra PV generated power.

In the past, linear search methods such as linear programming (LP), mixed-integer linear programming, etc., have been used to perform power scheduling efficiently in less complicated spaces [13], [14]. However, such methods are limited in handling stochasticity. Global optimization techniques such as genetic algorithm (GA), particle swarm optimization (PSO), etc., converge better than linear optimization algorithms due to their ability to handle stochastic system variables well [13], [14]. However, they are generally slow and are difficult to implement for online operation without hybridization with faster numerical techniques. This study proposes to use reinforcement learning to perform the scheduling task.

1.3 Reinforcement Learning

Reinforcement learning (RL) is one of the machine learning algorithms that is motivated by the behavioral adaptation of animals in their environment. It is built on the premise that animals experientially learn to optimize their behaviors in an environment. The goal of the learning process is to maximize the benefits in the environment as well as to avoid threats to their life in their habitat. Thus, it is a reward-motivated learning scheme [12].

To bring this solution scheme into power systems control, the scheduling problem must be expressed as a Markov Decision Process (MDP). An MDP is a formal mathematical structure for describing RL environments in order to enable the modelling of an RL agent to experience it in the way animals do with their natural habitat [11]. The Power scheduling in grid-connected microgrids may also be modeled as an MDP and RL be applied in solving it.

The advantage of the RL over other power scheduling algorithms is its ability to adapt to the optimization environment without an accurate model of the environment. Also, it can be deployed to schedule power in both a static and dynamic setting. In static scheduling, the RL is given a stationary state set over which it iterates to obtain optimal solutions over time. In dynamic scheduling, an RL agent is trained using a simulated environment and deployed to perform scheduling for a state set which it had not met during training [11], [15].

Recently, the application of RL techniques in power systems scheduling has attracted significant research interest. Jasmin [11] used a Q-learning algorithm to solve the classical power systems scheduling problems, i.e., economic dispatch, unit commitment and automatic generation control. In microgrid scheduling, RL has been applied in solving microgrid energy management by Mbuwir *et al* [16]. This research focuses on extending the emerging studies in the application of RL in microgrid power scheduling in the context of an EV charging station that is connected to the utility with incorporation of a battery storage system.

1.4 Research Motivation

Despite the emerging interest in the application of RL in microgrid power management, there are still outstanding challenges in its adoption in solving microgrid power scheduling.

First, the development of an MDP that accurately represents the real-time microgrid operations is quite challenging. There is no standard way of expressing the microgrid power scheduling problem as an MDP so that RL algorithms may be used solve it with reasonable accuracy. Indeed, conventional MDPs used to model microgrid power scheduling environments still produce sub-optimal results. This is because, either the action space is highly discretized, or the battery degradation cost is ignored in the design. Lithium-ion batteries are quite expensive; thus, the cost of battery degradation is an essential component of the microgrid scheduling task if the battery storage is involved.

Second, RL does not directly learn the cost minimization policy. The policy of cost minimization is inferred using a reward function, which the agent aims to maximize. The rewards are associated with the actions that the agent chooses in the action-space (search space). For example, in Q-learning, if the action-space is not properly constrained, the agent's learning process becomes unstable, thus fails to converge nicely to a global optimum. Convergence and stability are a real issue in the context of RL methods. Improving the global convergence and stability of RL methods is a subject that demands further research.

Third, Q-learning methods employ a lookup table to track learning. This limits their scalability and general performance in stochastic optimization. Recently, deep RL techniques such as the actor-critic methods have been developed that promise better performance and scalability. However, these algorithms still experience issues of instability, bias and variance in learning the

optimal policy. Improvement of the modern deep RL algorithms to enable them to offer better learning outcomes in power scheduling tasks would be of significant value.

1.5 Research Objectives

The main objective of this study is to develop and evaluate the performance of RL techniques for optimal power scheduling in a grid-connected PV/battery electric vehicle charging station. The cost considerations are the cost power purchase from the utility grid and battery degradation cost. The main objective of this research can be divided into the following sub-objectives:

i) To develop a mathematical optimization model for the power scheduling problem in a grid-tied PV/battery EV charging station including a battery degradation cost model.
ii) To develop a Markov Decision Process model to represent the operational environment of the EV charging station.
iii) To improve the stability and global cost convergence of a Q-learning algorithm in power scheduling for the EV charging station.
iv) To develop and evaluate the performance of an advantage actor-critic algorithm for power scheduling in the EV charging station.

1.6 Expected Outcomes of the Research

This dissertation proposes to modify the action space and the update procedure of a Q-learning algorithm to improve its convergence and stability. In doing this, the study develops a Q-learning algorithm that converges within fewer iterations to achieve higher cumulative rewards and a lower daily operational cost than the conventional Q-learning method.

Further, the study proposes an advantage actor-critic algorithm to solve power scheduling problem in the EV charging station. It is shown that this algorithm produces a lower daily operational cost than the conventional actor-critic technique under static conditions. Furthermore, it is shown that the algorithm offers a better balance between bias and variance in the learning process.

1.7 Limitations and Scope

This research deals with power scheduling on the consumer side of the utility grid. The consumer here is the owner of an EV charging station that is connected to the grid. Therefore, the algorithms

developed in this study are used to perform power scheduling in order to minimize the cost incurred by the consumer in supplying the load at the charging station. The load demand in this study is limited to the power requirements of the EVs getting to the charging station to recharge their battery packs.

Therefore, this study did not cover the voltage and current levels required for EV charging. Also, the voltage stability on the grid, i.e., the effect of the charging operation on other nodes in the power network, were ignored. It is assumed that the charging station operates within the instantaneous voltage, current and power limits set by the grid's distribution system operator. It is also, assumed that these limits are enough to maintain the required quality of service on the grid side. Moreover, the power electronic switching mechanism required to interface the charging station with the grid and the EV supply equipment is not within the scope of this study.

This study focuses only on power scheduling under static conditions; there was no considerations of dynamic conditions.

1.8 Dissertation outline

The rest of this dissertation is organized as follows.

Chapter 2 comprehensively reviews the various power scheduling algorithms that have been employed in the grid-tied microgrid schemes including RL techniques.

Chapter 3 describes reinforcement learning as a reward-based control algorithm. Each component of a general MDP is explained using a simple illustration of an RL environment. This is followed by an elaboration of the various reinforcement learning based control techniques used to solve MDPs.

Chapter 4 describes the development of an asynchronous Q-learning method to perform static power scheduling. Also, the chapter includes the description of the conventional Q-learning technique to benchmark the developed method.

Chapter 5 proposes a deep reinforcement learning method, specifically, an advantage actor-critic (A2C) algorithm to solve the power scheduling problem in the grid-connected charging station. Conventional actor-critic algorithm is also described for comparison purposes.

Chapter 6 provides the simulation set up, results and discussions. The four algorithms described in this study are implemented, tested. The algorithms are then compared in terms of their global cost convergence, stability and battery usage.

Chapter 7 gives the conclusions drawn from the results and the discussions as well as the recommendations for future research.

1.9 Research Outputs

As at the time of writing this dissertation, some of the outputs of this research had been presented in peer-reviewed local and international conferences. Also, two journal papers had been submitted for peer review. The outputs of the research are as follows.

Journal papers:

i) **Arwa O. Erick** and Komla A. Folly, "Improved Q-learning for Grid-tied PV Microgrid Energy Management," *submitted to SAIEE's Africa Research Journal, Special Issue 1, 2020 – under review*.

ii) **Arwa O. Erick** and Komla A. Folly, "Reinforcement Learning Approaches to Power Management in Grid-tied Microgrids: A Comprehensive Review," *submitted to IEEE Open Access Journal – under review.*

Conference papers:

i) **Arwa O. Erick** and Komla A. Folly, "Power Flow Management in Electric Vehicles Charging Station Using Reinforcement Learning," in the *2020 IEEE Congress on Evolutionary Computation (CEC*), Glasgow, United Kingdom, 2020, pp. 1-8, July 19-24, 2020.

ii) **Arwa O. Erick** and Komla A. Folly, "Power Flow Management in Multi-Source Electric Vehicle Charging Station," in the *21st International Federation of Automatic Control (IFAC) World Congress*, Berlin, Germany, July 11-17, 2020.

iii) **Arwa O. Erick** and Komla A. Folly, "Reinforcement Learning Approaches to Power Management in Grid-tied Microgrids: A Review," in *5th Clemson University Power Systems Conference,* Clemson, USA, March 10-13, 2020.

iv) **Arwa O. Erick** and Komla A. Folly, "Energy Trading in Grid-connected Electric Vehicle Charging Station," in *28th African Universities Power Engineering Conference,*

Mechatronics/Pattern Recognition Association (SAUPEC/RobMech/PRASA), Cape Town, South Africa, January 29-31, 2020.

2.0 A Review of Power Scheduling Techniques in Grid-Connected Microgrids

In this chapter, a comprehensive review of optimal power scheduling algorithms is provided. The review starts from the state-of-the-art of a grid-connected microgrid and the mathematical constructions for the power scheduling problem. This is followed by a review of the optimization techniques commonly used to solve the power flow problem including priority listing, linear search methods, global optimization algorithms, and reinforcement learning techniques. The chapter outlines the different challenges associated with conventional power optimization algorithms for grid-tied microgrid schemes. It also highlights the application of reinforcement learning methods to address the identified challenges.

2.1 Background

Optimal power flow (OPF) control involves the management of the power system variables to supply the load at minimum or reasonable cost while not violating the system constraints [17]. The main goals of this optimization are to minimize energy production and delivery costs, minimize power losses, reducing load shedding, and maximize system performance in general. The objective function captures the cost minimization while the constraints take care of the system health and load-generation balance [10], [17]. Particularly, optimal power scheduling encompasses the temporal arrangement of the system's power resources to achieve the objectives of the system's objective and maintain its overall health [11].

Traditionally, utility grids have had a one-way power flow, a power measurement scheme with a simplex information flow method and power generators whose outputs are more foreseeable [18]. The evolution of decentralized generators behind the meter systems (DGBMS) in the new smart grid model has introduced several challenges to the power systems all over the world [19], [20]. DGBMS is an electricity supply scheme where renewable energy generator (s) produces electrical power for on-site use [5].

Despite the advancement in research on renewable energy and smart grid control technologies, grid-tied microgrids still face challenges in the control of their operations to enable smooth energy cooperation with the utility grids [17]. The main challenges posed by the connection of DGBMS to the current grids include bidirectional power flows rendering classical protection systems

designed for unidirectional power flow obsolete, deleterious frequency distortions as a result of interaction between the distributed generator and the control systems and intermittent outputs necessitating the use of energy storage systems (ESS) [19], [20]. Furthermore, microgrid power dispatch and control is a challenging task due to the intermittency of load and some generation on both the microgrid and the utility grid side [10]. Studies have shown that grid-tied microgrids if properly controlled, can help improve the utility grid's resiliency, power quality and overall reliability [10]. Therefore, there is a need to employ robust dynamic control mechanisms to ensure smooth cooperation between the charging stations and the utility grid.

A grid-connected EV charging station can be viewed as a DGBMS dedicated to supplying the EV charging load as opposed to residential or industrial loads. With the increasing demand for fast charging stations, power flow control has become a major challenge as the electrical grid may not support the power requirements for high charging rates [21]. Besides, it has been proposed that the use of EVs instead of internal combustion engine vehicles (ICEVs) may be a viable solution for environmentally destructive greenhouse gas (GHG) emissions from the transportation industry [22]. However, the current grids are predominantly fossil fuel-powered, thus supply the EVs with unclean energy from fossil fuels cancels the environmental advantages associated with the EVs [23]. Therefore, there is a need to incorporate renewable energy sources (RES) to maximize the environmental benefits of electromobility.

Thus, the goal of optimal power scheduling algorithms in an EV charging station is to manage power so that the EV charging load is met at minimum cost [5], [21]. For a charging station supplied by both RES and the grid with a battery storage system integrated, the objective of the optimization is to decide on how much power should be drawn from each of the station's power sources in order to supply the station's load at minimum cost [21]. The costs involved are the cost of purchasing energy from the utility as well as the cost of degradation of the battery [18]. Incorporation of battery degradation costs in the optimization has been shown to extend the battery life because the control algorithms are designed to avoid an unnecessarily expensive charge or discharge operations [5].

2.2 Power Scheduling Problem Definition

2.2.1 Setup of a Grid-Connected PV/battery EV Charging Station

A typical grid-tied, solar-powered EV fast-charging station consists of a solar PV generator, and a battery storage system. These and the utility grid are linked via the necessary power conversion gear. This architecture is borrowed from conventional grid-tied microgrid systems. There are two major architectures for the microgrid systems, namely, the DC and the AC microgrids. The DC microgrids mainly use the DC bus system while the AC bus system is found in AC microgrids [24].

The decision on the bus system is based on the need to reduce the number of power conversion stages required to supply the loads with minimum power electronic converter losses. In EV charging stations, a common bus is vital to facilitate power sharing among the electric vehicle supply equipment (EVSE) [25]. The DC bus system has a mutual DC bus that is maintained at a steady DC voltage. A power converter links all other power supplies to this bus. The grid is connected to the DC bus via an AC to DC converter, while the solar PV generator is linked to the DC bus by a maximum power point tracking (MPPT) enabled DC to DC converter. MPPT control may be performed using various algorithms such as incremental conductance perturbation, fuzzy logic, heuristic techniques, perturb and observe algorithms, etc. [26]. This system is preferred due to less power conversion phases needed to deliver power to the electric vehicle. The EVSE may have just one DC to DC boost converter for all the EV chargers through a DC bus link [24].

Notwithstanding the choice of the bus system, the power conversion system for an EVSE needs to have enough switching speed to maintain the dynamic operations of the control system employed. Also, the conversion should be efficient to avoid power losses during dispatch. Fig. 2 shows the DC and AC bus systems architectures.

The station supplies an EV charging load. EV charging load is stochastic and statistical models have been developed to represent the vehicle arrival and departure characteristics for the optimal design of the charging stations [27], [28], [29].

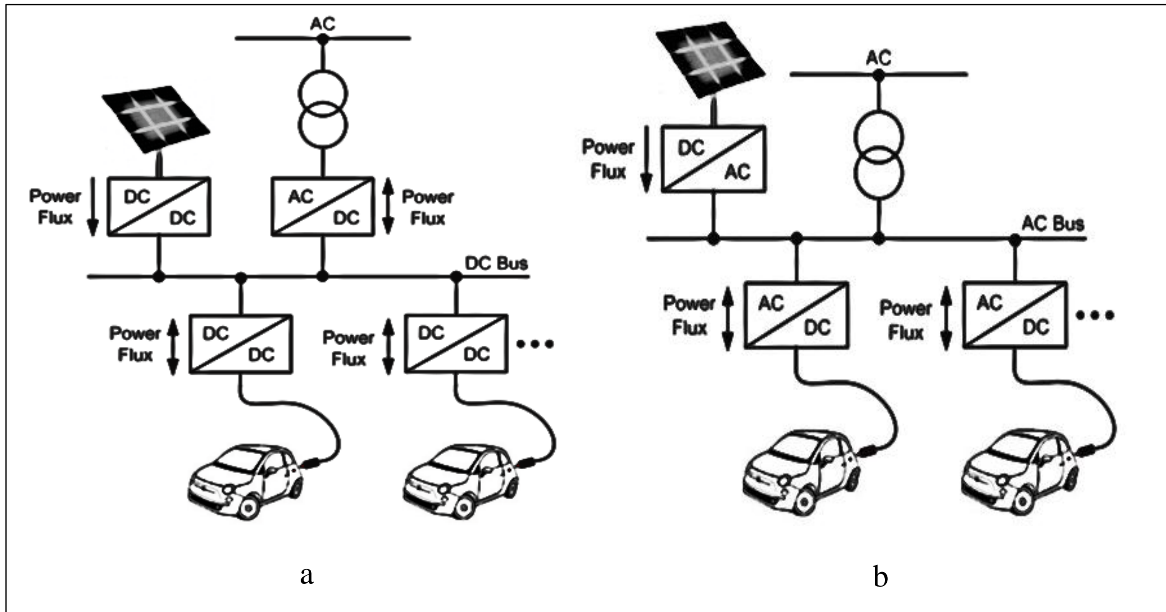

Fig. 2.1: Bus system configurations for an EV charging station: (a) DC bus system (b) AC bus system [25]

2.2.2 Mathematical Formulations of the Power Management Problem

The purpose of optimal power control systems for the grid-tied microgrids (GT-MGs) is to minimize operational costs and to maximize income from the sale of energy while meeting the load on the microgrid side [17], [30]. For a DGBMS, the essential variables required to solve the problem include the power output from the DG, the microgrid's load profile, the instantaneous state of charge (SoC) of the battery storage system (BSS) and the day-ahead tariff profile of the utility [17]. The costs considered include the grid power purchase cost, the cost of degradation of the BSS as defined in and the cost of power purchase from auxiliary sources [18].

To account for the cost of purchasing the grid power, the tariff system used by the utility is to be considered. In some studies, the grid tariff system used is dynamic and stochastic as in [17], [21] while in other studies, the tariff is assumed constant like in [31], [32].

Different authors present different formulations for the power management problem. The most common construction is the optimal battery scheduling approach [18]. In this formulation, the only decision to be made at every time step is the battery state of charge for the next time step. Once this decision is made, it is used to decide how much power is to be supplied to or drawn from the

battery to actualize the control decision. Thereafter, a power equilibrium equation is determines the quantity of power to be supplied to or drawn from the grid based on the current RES generator output [17], [33]. A simple power balance equation for a grid-tied PV-battery EV charging station is given in (2.1) as [21]:

$$P_{cl}(t) = P_{bss}(t) + P_g(t) + P_{pv}(t) \tag{2.1}$$

where $P_{cl}(t)$, $P_{bss}(t)$, $P_g(t)$ and $P_{pv}(t)$ are the station's load, the battery power, the grid power and the PV output respectively. Thus, if $P_{cl}(t)$ and $P_{pv}(t)$ are known, a decision on $P_{bss}(t)$ by the optimization algorithm in every time step leaves $P_g(t)$ as the only unknown in the power equilibrium equation. The other mathematical formulation for the power flow control problem is the unit commitment approach. In this formulation, the solver decides on both $P_{bss}(t)$ and $P_g(t)$ in a tuple of two elements $\{P_{bss}(t), P_g(t)\}$ [34]. If there are other sources of power that are not intermittent, they may also be added to this tuple [31], [32].

Regardless of the formulation used, there is a need for a solver or an optimization algorithm to ensure optimal power schedules are obtained. The optimization techniques that have been used to solve the optimal power scheduling problem for grid-tied microgrids can be categorized as priority listing, linear search, global search and RL methods.

2.3 Priority Listing

The priority listing method involves ranking the possible solutions in order of their contribution to achieving the objective of the optimization process. The merit list is created by intuitively determining which solution best minimizes the cost function based on simple pre-set rules and constraints. This method is the most basic approach to power systems optimization such as unit commitment [35]. The method is fast and can be run in a real-time scenario.

Abronzini *et al.* [8] developed an iterative priority list-based energy control system to optimize charging operation in an EV charging station taking into account multiple energy sources, namely, solar PV, the utility grid, battery ESS and V2G. The authors accounted for the cost of battery degradation, grid power import, PV power generation, vehicle-to-grid power import and efficiencies of power converters. The authors used the unit commitment formulation. They gave the highest priority to the cheapest power source starting from solar PV and the main grid to supply the EV charging load and energy injection into the grid. In [31], a priority listing method is applied

in optimal power control in multi-source EV charging. The main drawback of this approach is that it must be optimally initialized, otherwise it returns errors. For instance, an optimal initial battery state of charge was determined using a Monte Carlo simulation in [8].

2.4 Linear Search Methods

Linear techniques of optimization are generally used to find the optimum solution of linear, continuous and differentiable objective functions. Linear programming (LP) is one of the very computationally efficient algorithms for mathematical optimization [36]. Since the method only works with linear functions, the objective function must be linearized, if it is not linear. However, linearization comes with various assumptions that may render the solution sub-optimal. When used with the Model Predictive Control (MPC) paradigm, LP gives a robust optimization scheme for real-time controllers. An MPC controller uses the model of the system to predict its future behavior or state and solves an on-line optimization problem to select the best control action that produces the desired system trajectory. It is robust and stable as it has a feedback loop and uses the moving horizon technique [37]. Boo and Kim [38] implemented an MPC-LP based algorithm to optimally control power flow in an EV charging station. A simple MPC control algorithm showing the data flow in each of its entities is shown in Fig. 2.2. For non-convex objective functions, linear programming is infeasible.

Mixed-integer linear programming (MILP) can be used to successfully solve the power scheduling problem if at least one of the control variables is conditionally limited to an integer value. Zhang [39] discusses the application of MILP in optimal EV battery power control considering the battery state of health (SoH).

However, linear optimization methods have limited scalability and do not perform well in stochastic atmospheres like in RES-based GT-MGs. Besides, such simplex optimization algorithms are prone to being trapped in a local extremum which may result in sub-optimal solutions. Therefore, global search methods are normally applied in stochastic environments.

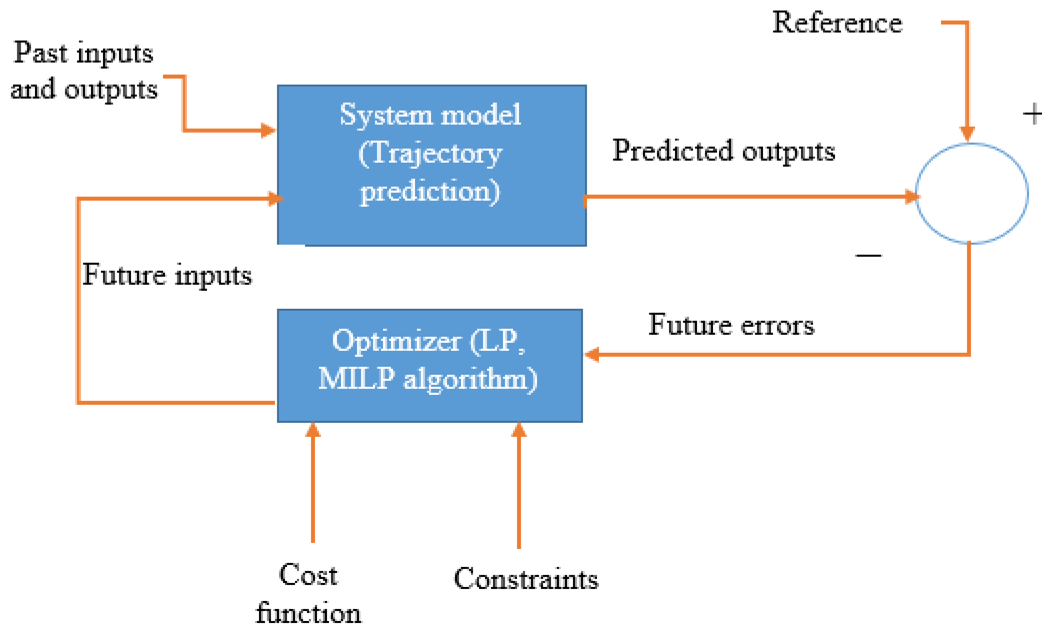

Fig. 2.2: Data flow diagram for an MPC algorithm [37]

2.5 Global Search Methods

Global search methods have the advantage of scalability and the ability to perform power control in highly stochastic settings. The major categories of global search methods include evolutionary algorithms and swarm intelligence.

2.5.1 Evolutionary Algorithms

Evolutionary algorithms are the optimization methods derived from the Darwinian theory of natural selection that stipulates that living organisms, by chance, acquire characteristics that make them fit or unfit for their environment [40]. There are several evolutionary algorithms such as genetic algorithm (GA), differential evolution (DE), etc. The most common of them is the genetic algorithm (GA). In a GA, a set of random solutions (population) is first initialized. Each member of the population is evaluated for fitness (using the objective function), after which changes in the population are implemented using crossing over, mutation and fertilization, processes which are all random [41]. The algorithm goes through several iterations (generations of the population) until the fitness function is optimized. El-Hendawi *et al.* [14] describe the use of a GA to optimize

power supply in a grid-connected PV-battery water pumping station. Hijjo *et al.,* [30] developed a GA-based algorithm for optimally designing and operating a PV-based microgrid connected to a diesel generator and the utility grid for backup. Specifically, the authors used the algorithm to obtain an optimum dispatch of PV generated power and the diesel considering the battery SoC constraints and the efficiencies of the converters.

2.5.2 **Swarm Intelligence**

Swarm intelligence is derived from the study of how swarms of insects or schools of fish search for food [40]. The most popular technique of this category is particle swarm optimization (PSO). PSO is reported to be more efficient and converges faster than GA [42]. Battery power scheduling using PSO starts with a set number of particles with positions represented initially by random values of battery energy levels or SoC throughout the optimization horizon, i.e., $SoC_1, SoC_2, \ldots, SoC_T$, where T is the optimization horizon. Another random set of SoC values is initialized to represent the swarm's initial global best solution. The power schedules resulting from each of the SoC sets are evaluated using the cost function. If the best cost among the particles is better than the global best, then that particle's position is set as the global best position and new particle positions are generated partly randomly and partly with respect to the particle's personal best and the current global best position. This iteration proceeds until the global best converges [13]. Tayab *et al.* [13] implemented a PSO-based procedure for power scheduling in a grid-connected PV-powered microgrid with a battery storage system (BSS).

Despite their ability to handle stochasticity in optimization, global search algorithms are generally slow and are incapable of operating online. This necessitates their hybridization with faster algorithms like dynamic programming (DP). In [43], a PSO-DP hybrid optimization system is designed to manage battery power levels in a grid-connected electric vehicle charging station with the integration of PV and BSS. The PSO is used to find the day-ahead optimal battery SoC for the charging station in steps of 1 hour. A DP algorithm with an error correction loop is implemented to control battery charge and discharge for each hour of operation in steps of 10 minutes. The error correction subroutine was implemented to deal with forecasting errors. Furthermore, these algorithms are developed for static optimization environments, hence their capacity for dynamic stochastic optimization is limited.

2.6 Reinforcement Learning Techniques

Reinforcement learning (RL) is a formal mathematical construction for solving sequential decision-making problems. In this category of algorithms, a software agent is modelled as a decision-maker that learns by iterative trial and error through a carefully defined reward scheme. The agent's key motivation is to maximize its total reward. For this learning to occur, the learning environment must be expressed as a Markov Decision Process (MDP). An MDP is the official language for describing sequential decision-making environments. An RL environment consists of states, actions, a reward function, and a state transition function [44].

In the optimal power scheduling problem, an agent can be trained using the various RL algorithms to take near-optimal real-time decisions on the flow of power in a grid-tied microgrid environment [18]. A state may be the current energy level of the battery, the grid tariff or the solar PV output or the microgrid's load, or a combination of all the four [21]. An action may be a decision on the battery instantaneous state of charge or energy or the power to be drawn from or supplied to the battery or the utility grid. With regard to this formulation, some researchers only model the actions to be three decisions, namely, charge, idle, or discharge as given in [15] and [45]. Action in the context of the unit commitment formulation may be a vector of power schedules for each unit as modelled in [34]. The action space is the union of all possible action sets for all the states in the environment.The reward is normally engineered to ensure that the objectives of the optimization process are in line with the agent's objective [18]. Fig. 2.3 shows a high-level illustration of an RL environment for optimal power scheduling in a PV-battery EV charging station.

The solution to the power scheduling problem begins with expressing the problem as an MDP. This includes the definition of the environment dynamics and the agent's behavior that best represents the microgrid's operations [18]. In microgrid schemes, the load, grid tariff, battery energy level or state of charge (SoC) and the RES generation are the possible state variables [21]. These variables may be modelled as continuous or discrete. In real-time, these values are continuous, thus discretization limits the accuracy of the environment's model thus rendering solutions that assume a perfect model inaccurate [44]. If all the information necessary to determine the state transition with certainty, then the environment is deterministic, i.e., the same action in the same state will always lead to the same next state [46]. Sometimes, very limited information is available in the current state and the agent is not fully certain about the next state.

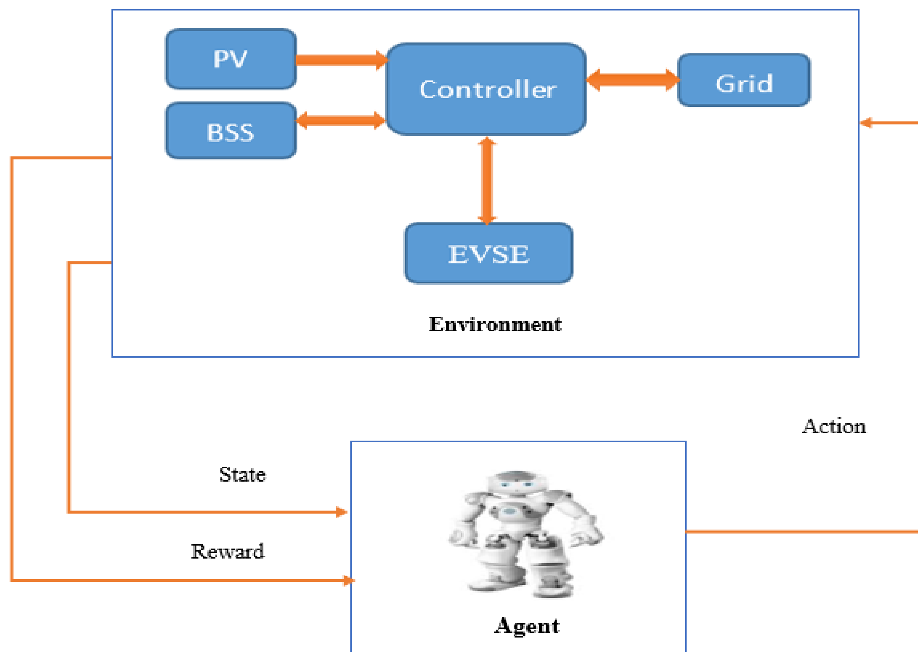

Fig. 2.3: An illustration of an agent-environment interaction for optimal power scheduling

The agent is designed use the little knowledge available to estimate the probability of the transiting to a given allowable state [47]. Such stochastic transitions are modelled using partially-observed MDPs (PO-MDPs) [48], [49]. Stochastic mathematical models have also been developed to represent load demand in EV CSs using the "*Spatio-temporal characteristics*" of EV location and arrivals at the EV charging stations [27]. Such models may be essential in approximating the state transitions in an EV charging station environment.

Under this construction, the agent learns by visiting states repetitiously and in a sequential manner, selects an action and proceeds to the next state and receives a reward as a result of the transition. It is in tracking the record of these transitions and updating the agent's learning experience that various RL algorithms differ.

The advantage of RL-based algorithms is that they can be trained to generalize a policy that can be used to solve similar problems even if the state set was not the one used in the training. Also, though training of an RL agent takes long, the training can occur offline and the learned policy retrieved for online operations [11]. Furthermore, the use of ANNs in deep RL algorithms eliminates the need for a separate forecasting model in the control algorithm which characterizes all other power scheduling algorithms [50], [49].

Several RL techniques have been applied in solving the power scheduling problem in GT-MGs, namely, dynamic programming, Q-learning, batch reinforcement learning, deep Q-network and policy gradient algorithms.

2.6.1 **Dynamic Programming**

Dynamic programming (DP) is the oldest of all reinforcement learning algorithms. It has conventionally been applied in determination of optimal policies by means of value or policy iteration in perfectly Markovian, i.e., the current state is all that is needed to get to the next state and not a sequence of the preceding states [44], [51]. DP is relatively efficient in solving small scale problems. Also, due to its feedback mechanism, it is easy to implement online error correction if the forecasting algorithm veers from the real measurements [43], [51].

In DGBMS, DP may be employed to determine optimal battery power for a day-ahead scheduling task. The task may be formulated as a problem of finding an optimal path, where each time step is marked with several nodes of battery SoC values. The agent determines which of the nodes of SoC values in each time-step is optimal. The SoC values are used to determine power values to be scheduled for each corresponding time-step. Tran *et al.* [52] developed a DP-based algorithm to schedule battery SoC in an islanded microgrid with a diesel generator backup. The DP algorithms was used to minimize the fuel cost, to maintain the battery state of health, and to maximize the profit from energy sales. Jeddi *et al.* [53] described a DP algorithm for optimally controlling battery charge and discharge in a grid-connected PV-powered home.

Nonetheless, dynamic programming has significant limitations. First, it assumes a perfect model of the environment that adversely affects the accuracy of solutions obtained through it. Second, DP-based methods are not scalable, thus may not only require large discretization but also be computationally expensive when applied to large scale systems [44], [54].

2.6.2 **Q-learning**

Developed by Christopher Watkins in 1989 [55], Q-learning is one of the most popularly applied reinforcement learning algorithms in the microgrid power scheduling sphere. It is an off-policy method that imposes more relaxed computational burden than on-policy algorithms such as DP and Monte Carlo sampling [55]. It also comes with the advantage of simplicity and versatility, i.e., the Q-learning construction may be framed for a large variety of sequential decision-making problems.

Jasmin [11] investigated the application of Q-learning to the most common power systems scheduling problems such as unit commitment, economic dispatch and automatic generation control. Q-learning application in grid-connected microgrid optimal power control is a mature concept in the available literature. Kuznetsova *et al.* [56] executed a 2-step Q-learning procedure to perform power scheduling in a grid-tied microgrid with a wind turbine based DG and BSS integration. A comparable construction is used with a three-step Q-learning to obtain a schedule BSS power in a grid-connected PV-BSS arrangement in [57]. Foruzan *et al.* [58] came up with a multi-agent-based Q-learning algorithm to control energy exchange amongst clients and energy providers, namely, the grid operators, diesel and wind turbine-based generators. In the aforementioned cases, the battery scheduling formulation of the power control problem is used due to its memory efficiency and simplicity.

The application of a Q-table to track the learning process have fundamental limitations like those associated with dynamic programming. As the dimensions of the environments increase, the size of the Q-table becomes larger, therefore, the algorithm suffers from the curse of dimensionality leading to poor convergence [44], [59].

One of the methods of overcoming this curse of dimensionality is to use artificial neural networks (ANNs) to approximate the Q-values based on statistical regression. Therefore, instead of learning the action-value function $Q(x, a)$, the algorithm learns the parameterized function $Q(x, a, \theta)$. Thus, to get the optimal value of the Q function, the function approximator finds the parameter θ^* such that $Q(x, a, \theta^*)$ best estimates $Q^*(x, a)$ [60]. Then in every state, the action that maximizes the estimated value function returned by the ANN is selected during policy retrieval. However, as Sutton *et al.* [60] observed, this method does not lead to proper convergence and diverges in some cases like Q-iterations, because the optimum policy is stochastic rather than deterministic. Therefore, a very small change in the value function for a particular action may well prevent it from being chosen, even if the action itself is the optimal one for that state.

2.6.3 **Batch Reinforcement Learning**

One of the major challenges experienced with traditional Q-learning is that it is data inefficient, thus it does not produce robust performance with stochastic policies [61]. Also, very complex environments with high stochasticity like DGBMS come with complex time series which may

cause instability in training the ANNs [49]. To ensure stability and data efficiency, batch reinforcement learning (BRL) is employed.

Claessens *et al.,* [62] combined the BRL and ANN Q-function estimation to attain up to 60% reduction in the peak load in a grid-tied PV/wind system. Shi *et al.,* [63] used an echo state neural network (ESNN) with BRL to obtain a 71% reduction in cost of energy in a grid-connected PV/battery microgrid.

The other commonly used approach is to randomly sample the saved agent's transitions from the replay buffer instead of using an ANN directly. This is called the "experience replay" technique [64]. In this method, every transition is saved in a replay buffer or memory as a tuple of the transition elements, namely, state, action, next state and reward. Then updates on the Q function are done using a random sample (minibatch) from the replay buffer like in Monte Carlo learning. This is unlike TD learning where the Q-value update is done using the most recent transition tuple. This technique has been used by Mbuwir *et al.* [16] to optimally schedule battery usage in a PV/battery/grid set-up. Because of the robustness of this approach, most modern deep reinforcement learning techniques apply experience replay method in updating the weights of the deep neural networks.

Though BRL is data-efficient, its accuracy is heavily data-dependent, and the algorithm cannot learn policies that have not appeared in its learning history. Also, for BRL algorithms that use ANNs to estimate the Q function, the issue of instability is often observed since Q-learning type recursion is not true gradient descent. This is because it involves updating the parameters of an ANN-based on an error with respect to a target which itself depends on the very parameters [65], [66].

2.6.4 **Deep Q-network**

Deep Q-network (DQN) is a deep reinforcement learning algorithm that combines techniques of both supervised learning and reinforcement learning. The DQN framework incorporates the deep learning methods into Q-learning iterations using the experience replay method borrowed from the batch reinforcement learning technique [49].

In DQN, the current policy is obeyed over a series of episodes and the outcomes are saved in a replay buffer. A random sample of the experiences in the buffer is then taken and used to update the deep neural network (DNN) [67]. The DQN algorithm employs two neural networks, namely,

the prediction network and the target network. The prediction network ($Q(s, a;\ \theta)$) estimates the current Q-value, while the target network ($Q(s, a;\ \theta^{-}$) hosts the old parameters used to estimate the next Q-value [59]. The present parameters (θ) are updated using a random sample of transitions (batch) from the replay memory after a set number of episodes. Then, after a given number of prediction steps, the parameters of the prediction network are copied into the target network [68].

Since the experiences from the replay buffer are randomly sampled, their sequential occurrence during learning which makes them correlated is interrupted. As such, the experience replay technique breaks the correlation between sequentially observed experiences thus reducing oscillations or even possible divergence of the action values returned by the Q-network. Besides, the use of a separate target network from the prediction network helps to achieve stability as opposed to previous methods where the same network was used.

Ji *et al.,* [34] describe the use of the DQN method to schedule microgrid energy generation and consumption with the inclusion of demand and generation prediction. The authors found that DQN returned a 20.75% reduction in energy cost compared to 13.12% obtained using a fitted Q-iteration. François-Lavet *et al.,* [45] implemented a DQN-based algorithm to schedule BSS and hydrogen storage for a microgrid in which they used convolutional neural networks to learn a general policy for scheduling the storage under unpredictable demand and generation environment. Lu *et al.,* [69] used the DQN strategy for energy trading between a microgrid and a power plant and achieved a 22. 3% improvement in self-consumption of the MG generated power.

2.6.5 **Policy Gradients Methods**

Policy is a way of acting. An ANN can be modeled as an agent to directly approximate this policy through policy gradient methods [66]. The policy gradient theorem is used to train an ANN to directly approximate the probability distribution with which action selection is done without the need for conventional action selection techniques. It has been established that learning a policy is easier and faster than learning a value function [66].

The actor-critic (AC) framework is the most popular policy gradient-based algorithm. Two deep neural networks are involved, namely, the actor and the critic networks. The actor approximates the policy and it is updated using a gradient ascent method. The critic network is a value function approximator which is updated through a gradient descent method. Generally, the AC algorithm

uses a stochastic policy. However, if the policy is modeled to be deterministic, the algorithm is called a deep deterministic policy gradient (DDPG) [59]. Also, recently, AC neural networks have been trained in parallel using an advantage function as opposed to the normal TD update method. An advantage function is expressed as the difference between the value of a state and the value of an action taken in that state [50]. The study on parallelization of AC learners led to the development of advantage actor-critic algorithms, namely, the asynchronous advantage actor-critic (A3C) and advantage actor-critic (A2C) algorithms [70]. It has been reported that these two algorithms display similar performance except that A3C is more complex to implement [71].

AC methods have attracted significant applications in GT-MGs power scheduling. Fuseli *et al.,* [72] employed an AC algorithm with a PSO-trained ANNs to obtain optimal schedules of energy resources in a smart home. The authors reported that, in terms of convergence, the algorithm performed better than the conventional PSO. In [73], an AC algorithm is employed to optimize power allocation in an assorted power network with wind turbines and PVs, leading to a significant increase in energy efficiency. Wan *et al.,* [74] scheduled energy storage in a smart home using an AC algorithm leading to a significant energy cost reduction. A3C has been used by Hua *et al.,* [75] to schedule battery energy storage for PV and wind sub-grids achieving better performance than conventional optimal power flow (OPF) solutions and improved utilization of the energy storage devices.

DDPG is known to perform better than AC with continuous action spaces [76]. Chen *et al* [77] used the DDPG algorithm to control energy flow in a grid-tied PV/battery microgrid. The algorithm returned a 55% improvement in earnings. Odonkor and Lewis [78] developed a DDPG-based controller for joint energy storage devices for building groups to obtain a decrease in the peak demand. Yu *et al.,* [79] designed and simulated a DDPG algorithm to manage energy storage and heating, ventilation and air conditioning (HVAC) devices attaining a reduction in energy cost by 10%. However, DDPG does not work with stochastic policies.

The other problem associated with such policy gradient-based algorithms is that they are incapable of taking more than one action at a time. Unlike DQN that returns the state-action value for a set of possible actions in a state, in deep policy gradient methods like DDPG and AC, the actor may only return one action or the probability of taking one action at a time in a particular state. Mocanu *et al.,* [80] developed a novel deep policy gradient (DPG) technique that was able to deal with more actions per state and converged faster than normal DDPG. Specifically, the authors used the

DPG algorithm for both energy cost minimization in HVAC devices and peak reduction for residential application of the devices. It was observed that this algorithm performed better than a DQN-based algorithm in realizing the peak demand reduction and optimization of energy use.

2.7 Summary

Q-learning is an efficient and simple to implement reinforcement learning algorithm. However, Q-learning type recursions come with instabilities that may cause poor convergence and inaccurate results. Q-learning forms the basis of the development of most other reinforcement learning algorithms. Therefore, the improvement of Q-learning to achieve better and stable convergence is worth investigating.

Moreover, the application of a Q-table as observed with traditional Q-learning is limited to small scale static optimization problems. In for large scale and dynamic problems, Q-learning is infeasible because it suffers from the curse of dimensionality. Furthermore, is not possible to correctly predict all the possible states and enumerate them in a Q-table. Deep RL methods such as the DQN and actor-critic algorithms can be used to overcome the lack of scalability. DQN is limited to just a small discrete action space. Furthermore, it is a pure value-based method that does not converge properly, and it is unstable. The actor-critic architecture is more robust than a pure value-based algorithm for power systems scheduling because of the following three reasons.

i) The algorithm applies the policy gradient theorem. It has been observed from reviewed literature that it is much easier to learn a policy than a value function. This implies that the algorithm converges faster than value-based reinforcement learning methods such as Q-learning and DQN.
ii) Actor-critic methods separate policy and value function models thus are more stable than other reinforcement learning algorithms such as Q-learning and DQN.
iii) The algorithms have the feedback mechanism that allows for seamless online error correction and adaptability. The implication of this capability is that it is possible to use the actor-critic algorithms to develop intelligent power scheduling systems which are capable of autonomous stochastic optimization.

In this dissertation, actor-critic algorithms are explored. Particularly, an advantage actor-critic (A2C) algorithm is developed. Modern deep reinforcement learning techniques such as experience

replay and parallelization of learning agents are used to improve the performance of the actor-critic architecture in static optimal power scheduling.

3.0 Reinforcement Learning: A Reward-Based Control Process

In this chapter, an introduction to reinforcement learning as a reward-based control algorithm is provided. The basic characteristics of a reinforcement learning environment are discussed. The various learning methods used to solve problems in this construction are highlighted. Aspects of this chapter that are of greatest interest are the characteristics of a Markov process, the Q-learning algorithm, policy gradient theorem and the role of artificial neural networks in reinforcement learning techniques.

3.1 Background

Reinforcement learning is a reward-motivated iterative control algorithm derived from how animals obtain knowledge to live optimally in an environment that was initially unknown to them [44]. An animal interacts with its environment sequentially (from one state to another), observing each condition (state) of the environment and taking actions. Every time an action is taken, the environment responds by changing state and gives a signal (reward) to the animal on the effect (value) of taking the specific action in the given state. The signal can be in the form of a burn if the animal touched a hot rock. When the animal meets such a rock again, it recalls what it previously did and the effect of what it did, and those two are enough to inform its behavior [11].

It was established that if animals can learn to behave optimally in their environment by trial and error, and motivated by reward signals, then such a learning methodology could be transmitted to cognitive science and computing. This realization led to the birth of the branch of computing called artificial intelligence (AI) [55]. Machine learning is the branch of AI by which computers can be enabled to make correct decisions without being explicitly programmed. The computer here is the learning agent with its learning characteristics modeled in a software module which informs how it behaves with input data.

Several techniques have been developed to help machines make optimal decisions or predictions, namely, supervised learning, unsupervised learning and reinforcement learning. Supervised learning involves the agent being given inputs and sample outputs so that it can learn a function or a rule that maps inputs to their corresponding outputs. This requires that the agent is supplied with labelled data, that is, the input-output pairs have labels that help to link them. In unsupervised

learning, the agent is supplied with inputs and it must determine internal features of the input data set so that it can distinguish it from other data sets. Unsupervised learning occurs without the need for data labelling. A reinforcement learning agent learns from its own experiences of an environment. An experience is defined as a set consisting of a state, an action, the next state and a reward.

Fig. 3.1 illustrates the interaction between an agent and its environment. By seeking to maximize the total reward in a set of several experiences, the agent learns a policy that links every state to the optimal action. A very recent example of reinforcement learning in real life is the way children learn to use phones to do tasks such as playing video games.

The advantage of reinforcement learning is that we do not need a large database of experiences in order to train an agent. For a child to learn to play a video game, it only needs the phone with the video game installed. The child will gather experiences and develop incredible expertise in playing the game. Furthermore, using this paradigm, the agent can be trained to solve complex problems without the need to stipulate the mechanisms of solving the problem itself to the agent [11]. In the recent times, reinforcement learning agents have been trained to outdo human experts in playing popular computer games [67].

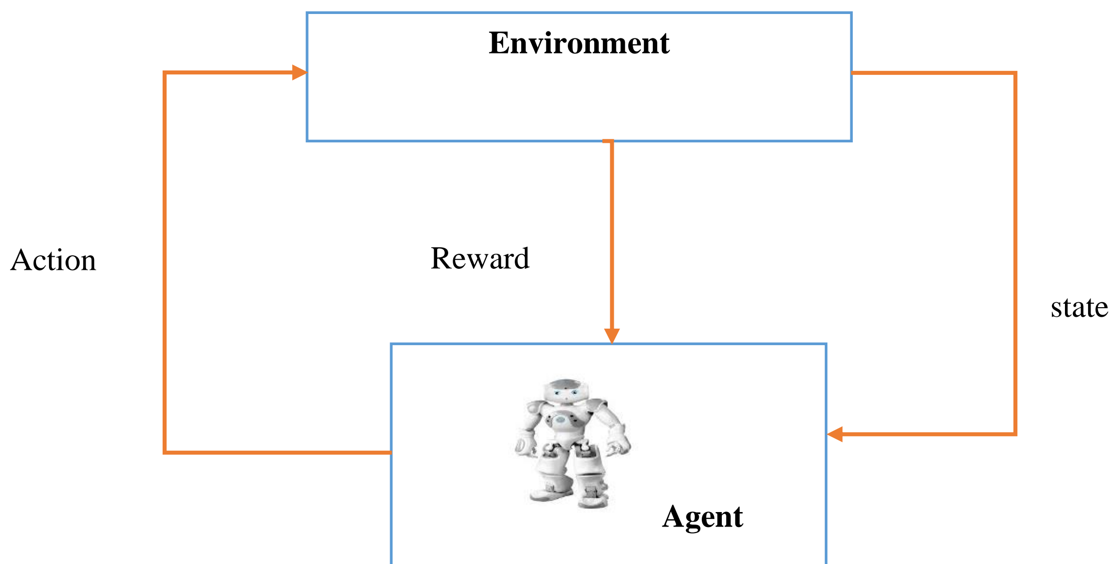

Fig. 3.1: An illustration of an RL agent-environment interaction

To bring the natural learning to computing and control theory, the problems to be solved must be defined in a manner that mimics the behavior of the environment, i.e., sequentially occurring states,

a set of possible (allowable) actions and a function that indicates how good it is to take any action in a particular state.

3.2 Markov Decision Process and Components of Reinforcement Learning

Since reinforcement learning is an interactive learning process that requires that an agent intermingles with an environment, how then must this learning process be formalized to enable the agent's experience to converge to an optimal policy? In 1957, Bellman [54] studied sequential decision-making processes and established the Markov Decision Process (MDP) model of multi-stage decision-making environments.

An MDP can be modeled by defining a environment with a state space in which a set of actions may be chosen and executed to occasion state transitions. The environment should have a fixed state transition model that changes the state of the environment once an action is taken in a given state [54]. Also, the MDP has fixed reward function that appropriates the value of the action taken. The reward function is supposed to be an approximation of the objective of the agent during the learning process. Sutton and Barto [44] described the role of an MDP construction in reinforcement learning as:

> "*…a considerable abstraction of the problem of goal-directed learning from interaction. It proposes that whatever the details of the sensory, memory, and control apparatus and whatever objective one is trying to achieve, any problem of learning goal-directed behavior can be reduced to three signals passing back and forth between an agent and its environment: one signal to represent the choices made by the agent (the actions), one signal to represent the basis on which the choices are made (the states), and one signal to define the agent's goal (the rewards).*"

Therefore, an MDP is the formal mathematical construction for sequential decision-making problems. It includes the agent and the environment in which it learns [47]. It is usually expressed as a tuple of four elements, (S, A, P, R), where S is the state space, A is the defined action space, P is the state changeover probability and R is the immediate reward obtained in taking a given control action in state S. The process is said to be "Markovian" if the state space is such that the next state is determined using the current state variables and the current action, without the need for a

memory of the events that lead to the current state [47]. It is this condition that allows for the definition of an optimality criterion in the learning process [54].

As an example, let the environment be a three-lane, finite-distance road. This forms a finite MDP such that there are finite transitions in one episode. Let the agent be a robot navigating this road from an initial point x_0 to a terminal point x_{T-1} in discrete steps (for simplicity). Let there be obstacles on the way at different points as shown by the red crosses in Fig. 3.2 These may be static representations of on-coming vehicles.

3.2.1 State and State Space

A state represents all the information from an environment that the agent requires to make the optimal decision at any given stage of learning.

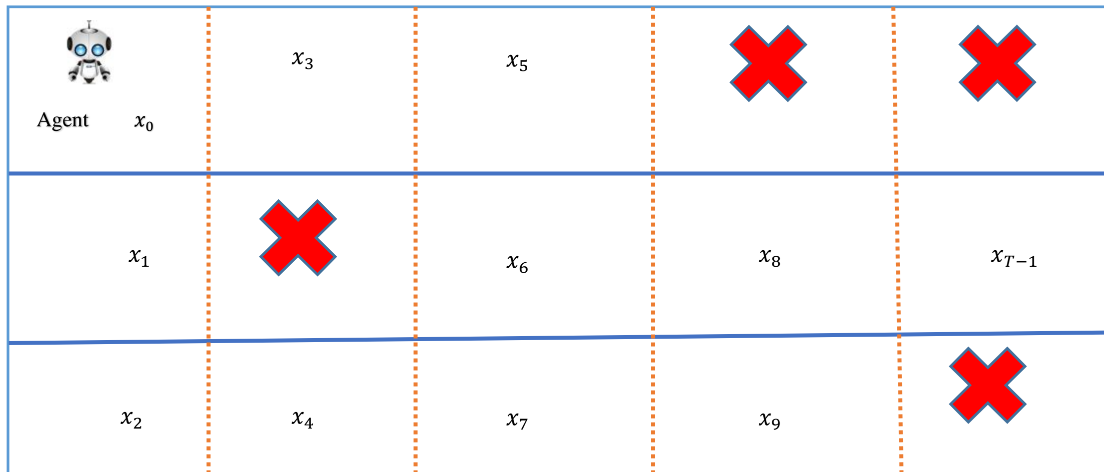

Fig. 3.2: High level illustration of an RL Environment

The state, in this case, is the position of the agent on the road labelled x_0 to x_{T-1}, where x_k represents all information about the environment at that position that the agent may require to make the correct decision on the movement. The combination of all possible states that an agent can be in during the learning process is the state space, i.e.,

$$\chi = x_0 \cup x_1 \cup, \dots, \cup x_{T-1} \tag{3.1}$$

Thus, for all time steps, $k = 0$ to $k = T - 1$, $x_k \in \chi$.

3.2.2 Action and Action Space

An action is a choice that the agent makes in each state. Every state has a specific action space from which an action may be selected. The action in a state can be to move left, right or front, and backward movement is disallowed. The agent is supposed to take a series of actions from the initial state x_0 to the terminal state x_{T-1}, that is, $a_0, a_1, a_2, \dots, a_{T-1}$.

The set of possible actions in a state is called the state-action space, $\mathcal{A}_k$. Each state has its own set of allowable actions. For example, at state x_3, the agent only has two options, namely, to move "left" or "front". The "left" movement leads to an obstacle. The union of all sets of all state action spaces is the overall action space.

$$\mathcal{A} = \mathcal{A}_0 \cup \mathcal{A}_1 \cup, \dots, \cup \mathcal{A}_{T-1} \tag{3.2}$$

3.2.3 State Transition

Depending on the model of the system, the state transition can be probabilistic or deterministic. In a deterministic environment, the same action in the same state will always lead to the same next state [55]. Therefore, the next state x_{k+1} only depends on the current state x_k and the action a_k, i.e.,

$$x_{k+1} = f(x_k, a_k) \tag{3.3}$$

However, in a PO-MDP, state transitions are probabilistic. Thus, such transitions are represented by a dynamic state transition matrix. The elements of this matrix are the transition probabilities of various possible next states [81]. For simplicity, a deterministic environment will be assumed.

3.2.4 Reward/Reinforcement Function

A reward is a real-valued scalar quantity that relays the objective of the learning activity to the agent. It is essential that the designer of the MDP intelligently chooses the reward function in order to achieve the purpose of the learning process [82]. The reward does not have to accurately represent the exact effect of the chosen action. An indication of progress in achieving the objective or lack thereof is sufficient.

An action is not only as good as the immediate reward it returns but also the state into which it leads [11]. A good action will produce high immediate rewards and lead to a state where there are potentially high rewards to be received. For example, being in a state with a non-zero probability

of hitting an obstacle such as x_5 is worse than being in a state where the probability of hitting an obstacle is zero such as x_6. Therefore, instead of rewarding an action, it is more correct to reward a transition. Thus, the reward is a function of the action taken in a state, the state in which the action is taken and the state it causes the agent to move to.

$$r(k) = g(x_k, a_k, x_{k+1}) \tag{3.4}$$

In the case of our 3-lane road environment, the goal of the agent may be to get to the terminal state using the fewest number of transitions and without hitting an obstacle. Let us assume that hitting an obstacle delays the journey but does not end it prematurely. To capture the objective, we can define the reward scheme as:

$$r(k) = -1 \text{ if } x_{k+1} \text{ is a normal transition}$$

$$r(k) = -5 \text{ if the agent hits an obstacle}$$

The total returns for an episode may then be calculated as:

$$\mathcal{R} = \sum_{k=0}^{T-1} [g(x_k, a_k, x_{k+1})] \tag{3.5}$$

3.2.5 Policy

A policy is a way of acting. It represents a rule by which actions must be taken in any given state. The goal of the agent is to acquire the knowledge of a policy that maximizes its total rewards. Since a single state may have several actions, a policy can be modelled mathematically as a probability distribution over the state's action space. The policy obeyed in a state x may be denoted by $\pi(x)$, and the optimal policy is denoted by $\pi^*(x)$. A policy is optimal in a particular state if it recommends the optimal action. Policies are compared using value functions to determine which of them is optimal.

3.2.6 Value function

There are two types of value functions, to be specific, state value function and state-action value function. The state value function is the measure of the goodness of being in a given state in view of the possible transitions that the agent can have beginning with that state. Value functions are defined based on particular ways of acting exhibited by the agent; thus, they are policy dependent

functions. To define the state value function, let us say the agent gets into a state x_k, chooses an action recommended by a given policy, π, and as a result, moves to the next state x_{k+1}. The value of the state x_k may be computed by Bellman equation [51]:

$$V^{\pi}(x_k) = r(\pi(x_k)) + \gamma \sum_{x_{k+1}} P(x_{k+1}|x_k)[\pi(x_k)]V^{\pi}(x_{k+1}). \tag{3.6}$$

where $r(\pi(x_k))$ is the immediate reward obtained in a state x_k by taking the action recommended by policy π, γ is the discount factor, $P(x_{k+1}|x_k)$ is the probability of getting to state x_{k+1} given the current state x_k, and $V^{\pi}(x_{k+1})$ is the value of state x_{k+1}. The discount factor determines how much of the expected future rewards are valued in the current state. Therefore, the value function is a good way of evaluating the goodness of a policy [11].

Since the value of a state is dependent on the subsequent states it leads to as a result of the policy being obeyed, Equation (3.6) can be extended to an infinite horizon MDP. We substitute the probabilities with an expectation operator so that [44],

$$V^{\pi}(x_k) = \mathbb{E}\left[\sum_{k=0}^{\infty}[\gamma^k r(\pi(x_k))]|x_0 = x_k\right]. \tag{3.7}$$

The goal of any learning algorithm is to converge at an optimal policy for all the given states. An MDP may have several optimal policies for different states. An optimal policy, π^* satisfies the condition:

$$\pi^* \in argmax_{\pi\in\Pi}V^{\pi}. \tag{3.8}$$

where Π is the ensemble of policies under consideration and the $argmax$ operator returns the policy for which the value, V^{π}, is maximum in the ensemble.

A state-action value function is defined as the quality of an action in a given state. If an agent takes an action a_k in a state x_k, moving to a state x_{k+1}, thus following a policy π, the state-action value function, $Q(x, a)$ is related to $V^{\pi}(x)$ as:

$$Q^{\pi}(x_k, a_k) = r_x(x_k, a_k) + \gamma \sum_{x_{k+1}} P(x_{k+1}|x_k)V^{\pi}(x+1). \tag{3.9}$$

The optimal Q-value, Q^*, for a state x_k if action a_k is taken following the optimal policy, π^*, is such that:

$$Q^*(x_k, a_k) = Q^{\pi^*}(x_k, a_k), \forall\, x_k \in \chi, \forall a_k \in \mathcal{A}. \quad (3.10)$$

If $Q^*(x_k, a_k)$ is known, then we can get the optimal policy by simply taking the action that maximizes this function. That is:

$$\pi^* = argmax_{\pi \in \Pi}\, Q^{\pi}(x_k, a_k), \forall\, x_k \in \chi, \forall a_k \in \mathcal{A} \quad (3.11)$$

And this is related to the optimal state value function according to the equation below:

$$V^*(x_k) = V^{\pi^*}(x_k) = \max_{a_k} \left\{ r(x_k, a_k) + \gamma \sum_{x_{k+1}} P(x_{k+1}|x_k)[\pi(x)]V^{\pi^*}(x_{k+1}) \right\}. \quad (3.12)$$

Or,

$$V^*(x_k) = \max_{a_k} \{ Q^*(x_k, a_k)\}. \quad (3.13)$$

Both value functions have been expressed in terms of policies. How then can the optimal policy be determined?

3.3 Solving Markov Decision Processes

Traditionally, MDPs are solved by applying the famous "Bellman's Principle of Optimality" which states that [51],

"An optimal policy has the property that, whatever the initial state and the initial decision are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decision."

This principle implies that if the value function for a state x_k is optimal under an optimal policy π, then if the policy is followed to the next state x_{k+1}, then the value of state x_{k+1} is also optimal. By this principle, optimal policies may be arrived at using two dynamic programming techniques, namely, value iteration and policy iteration.

3.3.1 Value Iteration

Value iteration is a cyclic estimate-and-improve process of determining an optimal state value function, thereby finding the optimal policy. The algorithm starts with random values and then iteratively adjusts the values until it converges to the optimal state value. Then using Equation (3.8), the optimal policy can be derived [44]. Value iteration is an off-policy method; thus, it executes with less computational resources than on-policy methods. However, it exhibits a slower convergence than on-policy methods.

3.3.2 Policy Iteration

Policy iteration starts with a random policy, then improves the policy at each time step until the process converges to an optimal policy. In each time step, the state value or state action value function is computed under the current policy to determine how good it is to follow that policy [11]. Policy iteration is more computationally expensive but converges faster than value iteration.

3.4 Reinforcement Learning Techniques

Reinforcement learning applies the concept of behavioral optimization exhibited by animals in an environment not known to them. It assumes that the learning agent is unaware of the state transition mechanism. However, through trial and error, an agent gathers experience tuples of state x_k, action a_k, reward $r(k)$ and next state x_{k+1}. The tuple $(x_k, a_k, r(k), x_{k+1})$ represents a single transition and it contains all the information necessary for an agent to learn and improve its behavior.

3.4.1 Monte Carlo Learning

Monte Carlo method is a powerful prediction and control tool. To perform prediction, Monte Carlo employs value iteration. Given a policy, $\pi(x)$, the prediction problem is to find $V^{\pi}(x)$. This is done by playing an episode, say n times. Let the reward obtained from the state x in an episode be $r(k)$. Therefore, the estimated value, $V^{\pi}(x)$, is the expectation value of $r(k)$ in the list of all the rewards throughout the episodes. That is,

$$V^{\pi}(x) = \mathbb{E}[r(k)] = \frac{1}{n}\sum_{k=0}^{n}[r(k)]. \tag{3.14}$$

However, in this study, the interest is performing optimal control. Optimal control is achieved using policy iteration with the Q-function. First, the Q-function is initialized with an arbitrary

value. Then the policy is set to random so that the agent takes a random action in each state in the state space. The policy is obeyed throughout an episode and all the rewards obtained are stored in a memory. The Q-value is then updated using the mean of all the values obtained in the episode. Equation (3.8) is then used to update the policy. This is done until the policy converges.

Monte Carlo control algorithm has some drawbacks. First, both π and Q start with random values and then π is used to find the value of Q. It is highly likely that π may recommend a wrong trajectory [44]. Also, learning, in this case, must wait until an episode ends.

3.4.2 Temporal Difference Learning

Temporal difference (TD) learning applies Bellman's Equation (3.6) to recursively update estimates of value functions of states or state-action pairs that are one time-step apart. The TD update is given by,

$$V(x_k) \leftarrow V(x_k) + \alpha(r(k+1) + \gamma V((k+1) - V(x_k)) \qquad (3.15)$$

Equation (3.15) means that the current value, $V(x_k)$, is updated adding the weighted difference between the new estimate and the current value. This difference is called the TD error. The weighting factor, α, is referred to as the learning rate. This recursion is done iteratively until the value converges to the optimal one.

3.4.3 Q-learning

Q-learning is an example of a TD control algorithm. Instead of using the state value V, Q-learning uses the state action value function, $Q(x_k, a_k)$, to determine the optimal policy. If an agent in a state x_k takes an action a_k and thus moves to the next state x_{k+1} by a probability P, the Q-value for that transition is given by:

$$Q(x_k, a_k) = r(k) + \gamma \sum_{x_{k+1}} [P(x_{k+1}|x_k)[\pi(x)]Q^{\pi}(x_{k+1})] \qquad (3.16)$$

The optimal Q-value is determined using Equation (3.10). The optimal action in the state is then determined such that:

$$Q^*(x_k, a^*) > Q(x_k, a_i), \forall\ a_i \neq a^* \qquad (3.17)$$

If this action exists for every state, then the algorithm has a guarantee of converge.

To determine the optimal policy, Q-learning begins by initializing the Q-values for each state-action pair to arbitrary values say, zeros. In every episode, n, the agent visits each of the states x_k, with state-action space, $\mathcal{A}_k$. The agent chooses an action, a_k, using a given exploration mechanism. It then transits to the next state x_{k+1} and receiving an instant reward, $r = g(x_k, a_k, x_{k+1})$. Then the Q-values are updated as given in Equation (3.18) [83].

$$Q^{n+1}(x_k, a_k) \leftarrow Q^n(x_k, a_k) + \alpha\left[g(x_k, a_k, x_{k+1}) + \gamma max_{a_{k+1}} Q^n(x_{k+1}, a_{k+1}) - Q^n(x_k, a_k)\right] \tag{3.18}$$

where $\alpha \in (0,1)$ is the learning rate which controls the extent of the modification of Q-values, $Q^n(x_k, a_k)$ is the current Q-value, $Q^{n+1}(x_k, a_k)$ is the next Q-value while $\gamma \in (0,1)$ is the discount factor. The terms in the square brackets are used to calculate the TD error in the Q-function estimate. The sum of the first two terms in the square bracket are the new Q-function estimate (referred to as the target Q-value) while the third term is the previous estimate (predicted) value. If the present state is a final one in the episode, then there is no subsequent state. Thus, the Q-value is adjusted using Equation (3.19).

$$Q^{n+1}(x_k, a_k) \leftarrow Q^n(x_k, a_k) + \alpha[g(x_k, a_k, x_{k+1}) - Q^n(x_k, a_k)] \tag{3.19}$$

Watkins and Dayan [83] proved that if α is small enough, Q^n converges to Q^* after a sufficient number of iterations. The number of iterations should be so much that each action is tried enough number of times for the Q-values to be correctly estimated using the rewards obtained. The Q-learning algorithm proceeds as shown in Fig. 3.3.

The Q-table can be created as a matrix (or a 2-dimensional array) with rows representing state indices and columns representing action indices. Each cell in the table represents a value $Q(x_k, a_k)$. This value is initialized by an arbitrary number, say, zero, for all the cells. Each time an action a_k is taken in a corresponding state x_k, the Q-value for the particular pair of state and action is updated according to Equation (3.18) or (3.19) depending on whether the state is the end of an episode.

The learning agent starts from the initial state x_0 and proceeds until the terminal state x_{T-1}. The trajectory taken informs the total rewards. There is an optimal trajectory from x_0 to x_{T-1}that returns the highest reward. The agent's goal is to find this optimal trajectory. But how does the agent take an action when it gets to a state? Does the agent take actions that it had tried and known the resultant rewards (exploitation) or try other actions to see if it can get better rewards

(exploration)? The solution to this exploration-exploitation dilemma is the objective of action selection strategies. The balance of exploration with exploitation is a very important part of reinforcement learning which differentiates it from other machine learning algorithms [44].

1. *Begin*
2. *Create Q-table*
3. $Q(x_k, a_k) = 0, \forall\, x \in \chi, \forall\, a \in \mathcal{A}, k = 0, 1, 2 \ldots, T - 1$
4. *Do number of episode times:*
5. *Read the current state*
6. *Do for each state:*
7. *Select action*
8. *Execute action*
9. *Get next state*
10. *Compute reward*
11. *Update* $Q(x_k, a_k)$
12. *End do*
13. *End do*
14. *Return Q-table*
15. *End*

Fig. 3.3:Q-learning algorithm [83]

3.4.4 Action Selection Strategies

The exploration-exploitation conundrum poses a challenge because the agent needs to accumulate a lot of experiences to take optimal action, but to gather that much data requires that a lot of computational resources must be expended. The action that gives the most reward is the greedy action, a_g. Strategies for solving the dilemma of exploration and exploitation have been developed, specifically, *ϵ-greedy algorithm*[11], *deterministic method* [56], *SoftMax function* [64], the *pursuit algorithm* [84], and the *random method* [85]. In the ϵ-greedy method, the greedy action is chosen by a probability, 1- ϵ, for $\epsilon \in (0,1)$, in any state x_k. Any other action in the action space, $\mathcal{A}_k$, have a probability, ϵ, of being selected [11]. The value of ϵ is typically set with a value close to 1 at the beginning. While the learning proceeds, it is gradually reduced until it gets to some set minimum

value close to 0. That requires hand tuning of ϵ, which may be inaccurate. For the case of the pursuit algorithm, an action a_k in state x_k has the probability P_k of being selected. P_k is usually set at an arbitrary value for all actions. It is then updated in each episode as given in Equation (3.20).

$$P_x^{n+1}(a_k) = \begin{cases} P_x^n(a_k) + \beta[1 - P_x^n(a_k)], & a_k = a_g \\ P_x^n(a_k) - \beta P_x^n(a_k), & a_k \neq a_g \end{cases} \tag{3.20}$$

where $\beta \in (0, 1)$ is a constant and n is the episode number. Therefore, in every learning episode, the pursuit algorithm gradually increases the probability of the greedy action being chosen while slightly reducing the probability of choosing the rest of the actions. Thus, if β is very small, as the number of episodes increases, the probability of choosing a_g in every state will get close to unity while that of choosing all other actions will collapse to near zero [86]. In the SoftMax method, every action has a probability of being chosen expressed as a function of its Q-value using Gibb's distribution function as described in [87]. The probability function is given by:

$$p(a|s) = \frac{exp\left\{\frac{Q(s,a)}{\tau}\right\}}{exp\left\{\sum_{a_g \in \mathcal{A}} \frac{Q(s,a')}{\tau}\right\}} \tag{3.21}$$

where $\tau \in (0,1)$ is the temperature. A small value of τ raises the probability of selecting a_g while big value makes selecting any of the actions in the action space equiprobable. Tokik [87] hybridized ϵ-greedy with the softmax method and came up with a more adaptive method of exploration called value difference based exploration (VDBE) where the value of ϵ was represented as a function of temporal difference error (TD-error) and he showed that the hybrid method is more robust than the ϵ-greedy and softmax methods.

3.4.5 Fitted Q-iteration: Q-learning with Function Approximation

In the previous section, Q-function updates were saved in a look-up table, called the Q-table. As state-action pairs increase, the Q-table size also increases, thus, the process suffers from the curse of dimensionality just like dynamic programming methods [18].

To overcome the curse of dimensionality artificial neural networks (ANNs) are normally employed to estimate the Q-function based on statistical regression. There are several function approximation

methods such as decision trees and multivariable regression techniques. ANNs are chosen for reinforcement learning because of the following reasons [44]:

i) They can deal with time-varying target functions.
ii) They can learn effectively using data acquired by incremental experiences.

ANNs are inspired by the way the human brain processes information by relaying it from one neuron to another with each neuron representing a transformation. Deep learning techniques are built around neural networks as function approximators. A single neuron in an artificial neural network consists of three main parts, i.e., a sequence of links with weights that transform the input signal, a summation operator that adds the weighted signal and an activation function that reduces the output signal to a value within the required range [88].

Fig.3.4 shows a simple illustration of the structure of a neuron in an ANN. Let v_k be the output of the summation junction, so that [88], [50]:

$$v_k = \sum_{j=1}^{m} w_{kj} x_j \tag{3.22}$$

And,

$$y_k = \varphi(v_k + b_k) \tag{3.23}$$

Where $x_1, x_2, \ldots, x_m$ is an $m \times 1$ input matrix so that x_j is an input signal, w_{kj} is the weight of link *j* connected to neuron *k*, φ is the activation function and b_k is the bias signal at node *k*.

The activation function can be linear, binary step, sigmoid, rectified linear, or SoftMax function depending on the nature of the output needed [89]. Of interest in this dissertation are the rectified linear unit (ReLU) and SoftMax functions. ReLU is applied where only positive linear transformation is needed. The function collapses to zero all negative values and passes the positive values as they are. The ReLU activation is defined as:

$$f_{ReLU}(x) = \begin{cases} x, \forall\, x \geq 0 \\ 0, \forall\, x < 0 \end{cases} \tag{3.24}$$

The SoftMax function returns a probability distribution of the shape of the input. The sum of all the output values of the function is 1. It is defined by:

$$g(z)_j = \frac{exp\{z_j\}}{\sum_k exp\{z_k\}},\tag{3.25}$$

where $j = 1, 2, 3, \ldots, k$.

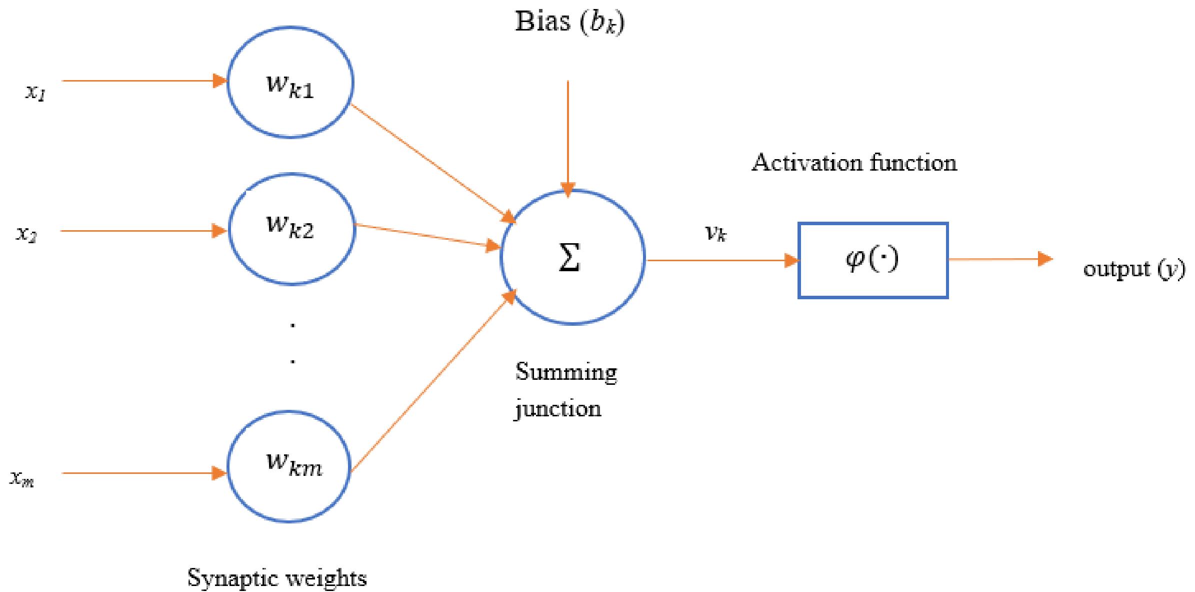

Fig. 3.4: Anatomy of a single neuron in an artificial neural network [88]

It is, therefore, possible to replace a function $f(x)$ with a parameterized function $f(x, \theta)$ where θ represents the weights vector of the neural network, w_{kj}. To estimate f, we only need the weights vector θ.

In fitted Q-iteration, instead of learning the action-value function $Q(x, a)$, the algorithm learns the parameterized function $Q(x, a, \theta)$. Thus, to get the optimal value of the Q function, the function approximator finds the parameter θ^* such that $Q(x, a, \theta^*)$ best estimates $Q^*(x, a)$ [60]. Then in every state, the action that maximizes the approximate value function returned by the ANN is selected during policy retrieval.

Many training algorithms have been developed to get the neural network to arrive at the optimal θ^*. The most common of them is backpropagation that uses the gradient descent technique to arrive at the optimal weight vector. The algorithm calculates the gradient of a loss function with respect

to each element in the weight vector $\theta^*$with the objective of minimizing the loss starting from the output layer backward [89].

In Q-learning, the Q-value of a state-action pair is normally updated by adding to the old value a TD error in the estimation of the Q-function. The new estimate of the Q-value is computed by adding the immediate reward to the maximum possible Q-value that may be found in the next state if the current policy were obeyed. In fitted Q-iteration, the new estimate is called the "target" (T_k^Q) and the old value is called the "prediction". Thus,

$$T_k^{Q^n} = g(x_k, a_k, x_{k+1}) + \gamma max_{a_{k+1}} Q(x_{k+1}, a_{k+1}, \theta_n) \tag{3.26}$$

In "stochastic gradient descent", the ANN is trained to minimize a squared error (loss) [50].

$$L(\theta) = \left(Q(x_k, a_k, \theta_n) \ - T_k^{Q^n}\right)^2 \tag{3.27}$$

Then the weights vector is adjusted using chain rule according to the stochastic gradient descent method [50]:

$$\theta_{n+1} = \theta_n + \alpha \left(Q(x_k, a_k, \theta_n) \ - T_k^{Q^n}\right) \nabla_{\theta_n} Q(x_k, a_k, \theta_n) \tag{3.28}$$

The update is done iteratively until the vector θ converges to θ^*.

However, as Sutton [60] observed, this method does not lead to proper convergence and diverges in some cases. This is because the optimum policy is stochastic rather than deterministic. Therefore, a very small change in the value function for a particular action may prevent it from being chosen, though the action itself is the optimal one for that state. Also, there are instabilities associated with Q-learning type recursion when it is applied to neural networks. The instabilities are caused by two issues:

i) Significant correlations within the state transitions. This is because the state transitions occur sequentially. Therefore, every state has some correlation with its predecessor and successor states.
ii) Fitted Q-iteration does not employ a true gradient descent. Instead, the algorithm updates the weights of a neural network based on a loss with respect to a target which also depends on the very weights [65], [66].

The first issue may be addressed using batch reinforcement learning.

3.4.6 **Batch Reinforcement Learning**

Batch reinforcement learning (BRL) applies the Q-learning type recursion technique. However, instead of updating Q-values each time an action is taken, it accumulates the agent's experiences of the environment in a replay buffer and uses a random batch of them to train an ANN (or other models) to estimate the Q-function [65]. Therefore, the agent approximates the optimal policy using a randomly picked batch of its earlier experiences in a method called batch gradient descent. Thus, BRL is said to converge quicker than conventional fitted Q-iteration and Q-learning that throws away transitions after each update of Q-values. The data flow diagram for a BRL algorithm is shown in Fig. 3.5 [65].

Storing transitions in a replay buffer and using random mini batches of the transitions to update the policy is called experience replay. Apart from breaking correlations within successive transitions, experience replay enables the retrieval of experiences that are more beneficial to the agent instead of discarding the transitions after each update as in traditional Q-learning. However, the application of experience replay alone does not completely solve the instability problem.

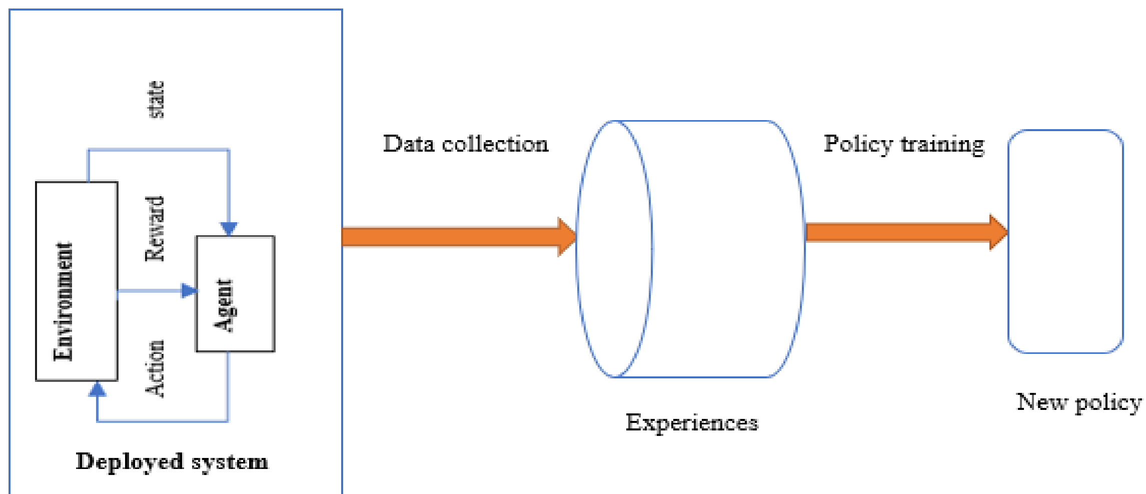

Fig. 3.5: Data flow diagram for a batch reinforcement learning algorithm [65]

3.4.7 **Deep Q-network**

Deep Q-network (DQN) was developed by Google's DeepMind Technologies to address both issues that cause instability in the fitted Q-iteration [49]. The DQN algorithm not only employs experience replay technique but also uses two different neural networks. One neural network

(policy or prediction network) estimates the current Q-value while the other estimates the target Q-value. The separation of the target network from the prediction network effectively reduces the instabilities generated by the neural network training process. The target network is just but a copy of the prediction network which is updated with the weights of the prediction network after every set number of steps.

To achieve the optimal policy, both prediction and target networks are initialized by random weights. The replay buffer capacity is also set to a fixed value. In every episode, the algorithm transitions through the entire state space. The current state is forwarded to the prediction network. The prediction network returns the estimated Q-values for all the possible actions. In each state, an action is selected, normally using the ϵ-greedy method. The action is executed, and the next state is computed. The reward is then calculated, after which the transition tuple is saved in the replay buffer as single unit: *(state, action, next state, reward).* After a given number of transitions, a random batch of the transitions is taken. A vector of current states in the minibatch is forwarded to the prediction network to get the predicted Q-values. The vector of all corresponding next states is forwarded to the target network to obtain the target Q-values. The squared loss vector between the target and the predicted Q-values is then computed and used to update the prediction network according to the stochastic gradient descent. After a predetermined number of steps, the weights of the prediction network are copied into the target network [90], [34]. The process continues until the objective is met.

3.4.8 **Policy Gradient Theorem**

The reinforcement learning methods discussed in previous sections use the value functions to estimate the policy. However, since the policy is the logic of an agent, an ANN can be modeled as an agent to directly approximate this policy through policy gradient methods [45]. It has been established that learning a policy in this manner is easier and converges faster than the use of value functions [60], [66]. The policy gradient theorem enables the direct parameterization of the policy as a probability distribution by which actions can be selected without the application of the action selection strategies discussed earlier.

The policy gradient theorem is defined as:

$$\frac{\partial \mathcal{P}}{\partial \theta} = \sum_{x} d^{\pi}(x) \sum_{a} \frac{\partial \pi(x,a)}{\partial \theta} Q^{\pi}(x,a), \tag{3.29}$$

where θ is the vector of the parameters of the current prevailing policy, i.e., weights of a neural network representing the policy, $\mathcal{P}$ is the average reward obtained if that policy is obeyed in every step and $d^{\pi}(x)$ is the fixed distribution of states under the prevailing policy. Since $Q^{\pi}(x,a)$ is unknown, it may also be estimated using a neural network or calculated by an N-step return, i.e., the total expected discounted reward for N stages (say taking 5 future states starting from the current one). This is because the absolute value of $Q^{\pi}(x,a)$ is not needed but how much better it is than the current policy [91]. An N-step return is defined as:

$$R_N(k) = r(k+1) + \gamma r(k+2) + \ldots + \gamma^{N-1} V(x(k+1)), \tag{3.30}$$

where s is the state, k is the step index.

Using Equation (3.29), the ANN can be trained by iteratively adjusting its parameters in the direction of a better policy performance using the gradient ascent method. Thus, the parameter θ is updated as follows [66]:

$$\theta_{n+1} = \theta_n + \alpha \frac{\partial \mathcal{P}}{\partial \theta_n}, \tag{3.31}$$

where α is the learning rate. The gist of policy gradient theorem is to express the policy as a function of the neural network weights (parameters) vector θ so that it can be differentiated as in the stochastic gradient descent to update the weights.

Let $J(\theta)$ be the objective of the learning agent expressed as a function of the parameter vector θ of the neural network agent, $\emptyset$ be a set of all possible state-action pairs representing a trajectory such that $\emptyset = \{(x_k, a_k), k = 1, 2, \ldots, T-1\}$ and R is the total expected discounted rewards from this trajectory, so that [66]:

$$R(\emptyset) = \sum_{k=0}^{T-1} r(x_k, a_k) \tag{3.32}$$

Then the objective of a learning agent is to maximize the expected return. Thus,

$$J(\theta) = \mathbb{E}[R(\emptyset);\ \pi(\theta)] \tag{3.33}$$

That is to say that the objective of the agent is expressed as the expected value of $R(\emptyset)$given the policy distribution $\pi(\theta)$expressed as a function of the weights matrix θ. The expectation operator allows for the consideration of all allowable trajectories the episode $\emptyset$ could have gone under the policy distribution. The policy gradient, therefore, requires the computation of the differential [91]:

$$\nabla_\theta J(\theta) = \nabla_\theta \mathbb{E}[R(\emptyset);\ \pi(\theta)] = \mathbb{E}[\nabla_\theta log\ (\pi(\theta))\ R(\emptyset)] \tag{3.34}$$

Since the ensemble of trajectories can be sampled, the policy neural network update is a stochastic gradient ascent process given by:

$$\theta_{n+1} = \theta_n + R(n+1)\alpha \nabla_\theta log\ (\pi(\theta)) \tag{3.35}$$

This means that the update increases the probabilities of selecting actions that give positive rewards and reduces the probabilities of selecting actions that return negative rewards. Otherwise, if all the rewards are positive, then the policy gradient update leads to a faster increase in the probability of picking actions with higher rewards. Also, the return $R(\emptyset)$ can be substituted by $Q(s_k, a_k)$ or the state value function and be estimated by another neural network.

3.4.9 **Actor-Critic Architectures**

The actor-critic algorithm is a hybrid technique that combines value function methods with policy gradients. The arrangement consists of two models, namely, the policy network and the value function network. The policy network (or the actor) takes the environment states as inputs and gives a control action (or a policy) as an output. The value function estimator (or the critic) observes the environment state and the reward obtained from the actor's control action and returns the estimated value of the state-action pair. The actor-critic (AC) data flow diagram is shown in Fig. 3.6.

If an actor takes (or recommends) an action a_k in a state s_k with a value $V(s_k)$, receiving a reward $r(k)$ and leading to a state s_{k+1} with a value $V(s_{k+1})$, the temporal difference (TD) error, $\partial(k)$, is computed as in Equation (3.36) [92], [93].

$$\partial_V(k) = r(k) + \gamma V(s_{k+1}) - V(s_k) \tag{3.36}$$

The TD error can also be computed using the state-action value function as [44]:

$$\partial_Q(k) = r(k) + \gamma max_{a_{k+1}} Q(x_{k+1}, a_{k+1}) - Q(x_k, a_k) \tag{3.37}$$

Furthermore, it is possible to get the difference between state-action value and the state value to find the numerical value of the advantage, $A(x_k, a_k)$, of taking a particular action in a particular state compared to any arbitrary action in that state [50]. Thus:

$$A(x_k, a_k) = Q(x_k, a_k) - V(s_k) \tag{3.38}$$

The actor is updated using stochastic gradient ascent to follow the direction of the policy that maximizes the total rewards, while the critic is updated using the stochastic gradient descent to the direction that minimizes the loss in the value function prediction.

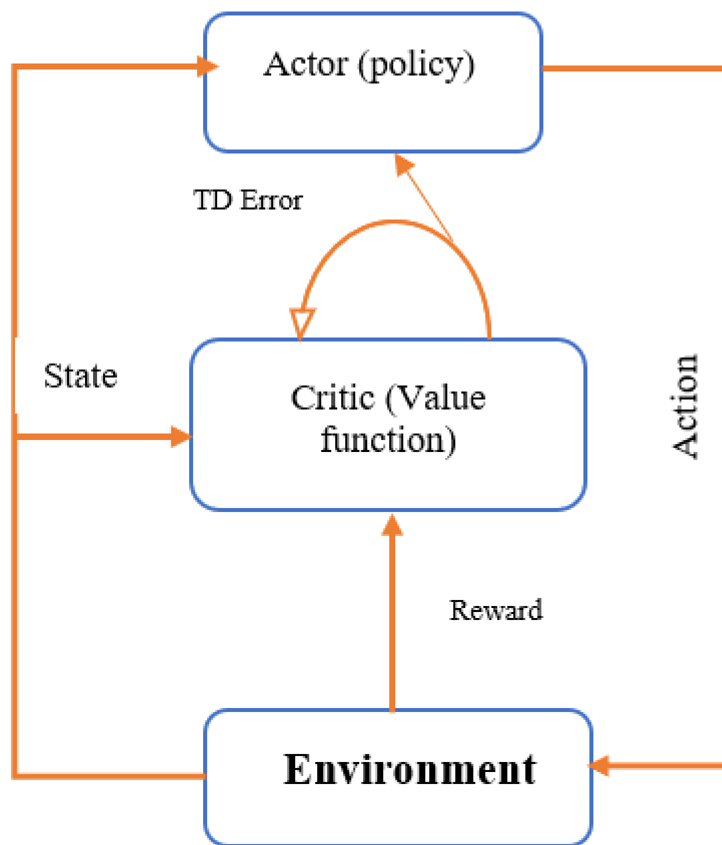

Fig. 3.6: Data flow diagram for an actor-critic algorithm [94]

3.5 Summary

Reinforcement learning has been described in this chapter. MDPs and their solution methods have been discussed. Important policy and value function update methods have also been highlighted. The policy gradient theorem and the update equations associated with it have been explained.

Furthermore, the actor-critic construction has been discussed. From their general design, the RL algorithms are very versatile. They may be used to solve most sequential decision-making problems. Power scheduling in a grid-tied PV-battery EV charging station is one of such problems. Therefore, provided we can define the operational environment of the charging station as an MDP, any of the RL algorithms may be used to solve it. The focus of the following chapter is the development the MDP. Also, an asynchronous Q-learning algorithm is designed to solve the power scheduling problem.

4.0 Q-Learning for Power Scheduling in Grid-Tied EV Charging Station

This chapter models the power scheduling problem in the EV charging station as a Markov Decision Process. It also proposes an asynchronous Q-learning algorithm to solve the power scheduling problem. The chapter starts by giving a step-by-step procedure for formulating the power scheduling problem. This includes the description of the model of the charging station, the formulation of the objective function and the description of the cost components such as the battery degradation cost and the utility grid tariff. This is followed by the development of a Markov Decision Process for the problem. Finally, an asynchronous Q-learning algorithm is proposed to solve the problem.

4.1 Background

Grid-tied renewable energy sources (RES) based electric vehicle charging stations are an example of a distributed generator behind meter systems (DGBMS). DGBMS are associated with several stochastic variables to be considered in each decision step when performing a day-ahead power scheduling [18]. These variables include the load profile of the station, the RES generator's day-ahead output profile and the utility grid's tariff profile [21]. Due to their learning component and ability to generalize solutions, reinforcement learning techniques are known to have the capacity to deal with stochastic problems more easily than most optimization methods [11], [71].

Q-learning is one of the most popular reinforcement learning methods due to its simplicity, versatility and guarantee of convergence [18], [55]. It employs temporal (TD) difference update rule, i.e., updating an estimated value with another estimated value. While this kind of recursion creates instabilities with deep learning models, it imposes less taxing computational burden than on-policy methods such as Monte Carlo and DP control algorithms [44], [55]. In [11], Q-learning is applied to perform the main utility grid scheduling problems such as economic dispatch, automatic generation control and unit commitment.

In grid-connected microgrid power scheduling, Q-learning has been used to obtain optimal day-ahead battery schedules. Kuznetsova *et al.* [56] implemented a two-step ahead Q-learning algorithm with a deterministic exploration method to schedule energy storage in a GT-MG with wind-powered DG, but with large discretization in both the battery energy and the wind power

generation. A similar model is used with a 3-step ahead learning by Leo *et al.* [57] to schedule a BSS in a grid-tied PV/battery system, where the authors reported an improvement in the utilization of the PV and the BSS. Foruzan *et al.* [58] developed a multi-agent scheme to manage energy trading between customers and energy suppliers including the utility grid, diesel and wind generators. Each entity trading with the grid was modelled as an agent, with each agent learning to improve on defined performance parameters, leading to a convergence that reduced the energy interchange with the grid by an average of 14%. The study in [21] investigates the application of Q-learning in managing energy cooperation between a PV power EV charging station with the utility grid. However, little research has been done on the application of Q-learning in a grid-connected EV charging station considering a dynamic tariff scheme. Furthermore, Q-learning techniques applied to microgrid power scheduling have issues of poor convergence and general instability in due to their high sensitivity to the learning rate.

This study develops an asynchronous Q-learning method to obtain optimal schedules for grid and battery power in a grid-connected electric vehicle (EV) charging station. The station is supplied by a PV power generator with a backup from the utility grid. The grid tariff model is dynamic in line with the smart grid paradigm. First, the mathematical formulation of the problem will be developed highlighting each of the cost components in the objective function. The formulation includes a battery degradation cost model. The problem will then be re-expressed as a Markov Decision Process (MDP), i.e., defining each of the parts of a reinforcement learning (RL) environment with regards to the power scheduling in the charging station. The MDP will then be solved using an asynchronous Q-learning algorithm. The conventional Q-learning based battery scheduling approach is also described for purposes of comparison.

Specifically, the study proposes to modify the action-space design of the algorithm so that each state has just the list of actions that meet the power balance constraint. By limiting the agent's actions in this manner, it is shown in this study that the modified algorithm returns a lower global cost and achieves higher total rewards than the conventional Q-learning method. Furthermore, the study shows that the asynchronous Q-learning method is more stable with regards to the sensitivity to the learning rate than the conventional Q-learning.

4.2 Mathematical Formulation of the Power Scheduling Problem

4.2.1 Charging Station Model

The charging station considered in this dissertation has a solar PV generator with a grid-tie supply and a battery energy storage system (BSS). The Solar PV generator, the grid and the battery are connected to the electric vehicle supply equipment (EVSE) via the appropriate power convertor as shown in Fig. 4.1.

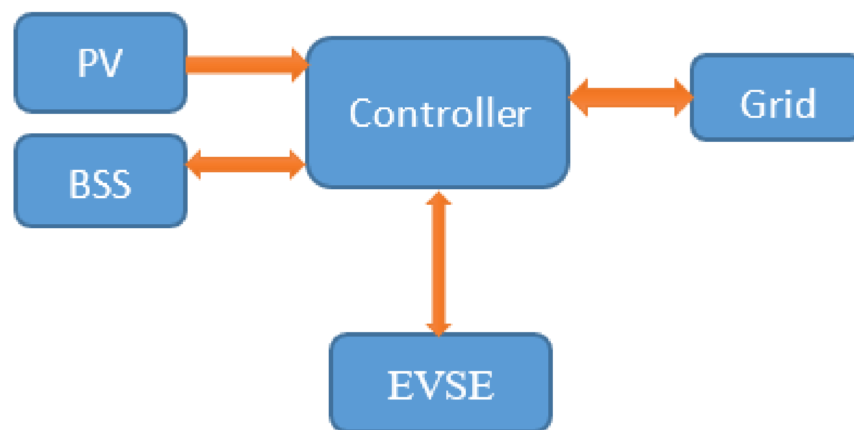

Fig. 4.1:Model of the EV charging station

A common bus is usually employed in EV charging to sustain the same power and voltage level in all the chargers that may be plugged into the bus [21], [95]. The DC bus system is preferred over AC bus system because it allows for power delivery to the EVs with fewer power conversion stages [25]. Also, fast charging is achieved at DC voltages according to IEEE Std 2030.1.1-2015, IEEE Standard Technical Specifications of a DC Quick Charger for Use with Electric Vehicles [96]. The DC bus system has a common DC bus maintained at a steady DC voltage level. Thus, one boost converter is enough to link the bus to the chargers [95]. All the loads and power sources are linked to it through an appropriate power conversion device. The grid is connected to the DC bus through a bidirectional AC-DC converter. The BSS is connected to the DC bus through a bidirectional DC-DC converter and the PV generator is linked to the DC bus through a unidirectional maximum power-point tracking (MPPT) enabled convertor.

4.2.2 **Objective Function**

The objective of the optimal scheduling algorithm is to minimize the operational cost of supplying a given 24-hour load profile while working within the operational constraints of the charging station. In this study, the cost of purchasing power from the grid and the degradation cost of the BSS have been taken into consideration. The main assumptions in the development of the cost function are as follows:

i) The power conversion in the power electronic interfaces is assumed to be lossless. Thus, the efficiency factor for all power conversion operations is assumed to be unity for simplicity.
ii) The charge and discharge operations are assumed to be lossless for simplicity. Thus, both charge and discharge efficiency factors are assumed to be 1.
iii) Power electronic conversion is assumed to be instant. There is no delay between the time a power value is recommended by the system controller and the time the power is delivered to the required component.

Consequently, the system variables required to solve the scheduling problem include the solar PV output power at every time-step, $P_{pv}(t)$, the instantaneous EV charging load at the station, $P_{cl}(t)$ the measured BSS instantaneous state of charge, $SoC(t)$ and the current grid tariff $G_t(t)$. The initial value of the battery state of charge, $SoC(t = 0)$, may be determined directly using indicators such as open-circuit voltage or estimated using various estimation methods such as Coulomb counting, support vector machines, artificial neural networks (ANNs), Kalman filtering any hybrid methods [97]. During the rest of the operational time, the state of charge dynamics for each time step, Δt, is described by [21]:

$$SoC(t) = SoC(t - \Delta t) - \frac{P_{bss}(t). \Delta t}{E_b} \tag{4.1}$$

where E_b and $P_{bss}(t)$ are the battery energy level at full charge and the battery power at the time respectively. The amount of energy in the battery at a time is given by:

$$E(t) = SoC(k)E_b \tag{4.2}$$

In each time step, the power equilibrium is governed by:

$$P_{cl}(t) = P_{bss}(t) + P_g(t) + P_{pv}(t), \tag{4.3}$$

where $P_{cl}(t)$, $P_{bss}(t)$, $P_g(t)$ and $P_{pv}(t)$ are the charging station's load, battery power, the grid power and the PV power at every time step . $P_{cl}(t)$ and $P_{pv}(t)$ are forecasted values. The algorithm is tasked with deciding the value of $P_{bss}(t)$ and $P_g(t)$. The energy cost, $C(t)$, in each time step is given by:

$$C(t) = C_{P_g}(t) + C_{P_{bss}}(t) \tag{4.4}$$

where $C_{P_g}(t)$ is the cost of buying power from the utility and $C_{P_{bss}}(t)$ is the cost of degradation of the BSS occasioned by continual charging and discharging. The battery power is taken to be positive when discharging and negative when charging. The cost of degradation is assumed to be the same in both charging and discharging for the same value of battery power. The grid power is positive when it is supplying power to meet the load at the station and negative when it is taking power from the charging station. For simplicity, the income obtained from selling energy to the utility has been ignored.

The objective of the optimizer is defined by the following non-linear and non-convex function:

$$Min\ (C_{tot}) = Min \sum_{t=0}^{T} \left[C_{P_g}(t) + C_{P_{bss}}(t) \right], \tag{4.5}$$

where T is the optimization horizon, i.e., 24-hours. The objective function is subject to the power balance equation and following other system constraints.

i) State of charge boundaries

The state of charge of the battery must be limited within pre-defined boundaries to protect it from overcharging and over-draining. The state of charge is limited by the following constraint:

$$SoC_{min} \leq SoC(t) \leq SoC_{max} \tag{4.6}$$

Where SoC_{min} and SoC_{max} are the upper and lower state of charge boundaries.

ii) Grid power limits

The power drawn from the utility grid must be within the allowable distribution transformer and transmission line limits. These limits may be set in a contract signed by the owners of the charging station with the managers of the utility. The grid power limits are given by:

$$P_g^{min} \leq P_g(t) \leq P_g^{max} \tag{4.7}$$

where $P_{g_{min}}$ and $P_{g_{max}}$ are minimum and maximum grid power supply to the station.

iii) BSS power

The power drawn from the batter is limited by the equation below:

$$P_{bss}^{min} \leq P_{bss}(t) \leq P_{bss}^{max} \tag{4.8}$$

The cost components of the objective function are explained as follows.

a) **Cost of power import from the grid**

To compute the cost of grid power purchase, there is a necessity to consider the tariffing system implemented by the operators of the utility grid. The tariff may be constant or varying depending on certain factors of production and power delivery. The factors include the load demand on the grid, fuel costs, the generation equipment availability and the amount of power that can be drawn from the available generators [98].

There are several types of tariffs which include flat tariffs, time-of-use (ToU) tariffs, real-time pricing (RTP), critical peak pricing (CPP) [98]. Flat tariffs have a time-invariant price for electricity supply. The ToU model is divided into four price times: flat period, peak period, valley period and pinnacle peak period, each with different electricity prices. Both RTP and CPP depend on the smart metering systems. In the RTP system, the tariff is dynamic but consumers are informed on the day-ahead tariff profile, while CPP is a hybrid of ToU structure and the RTP system [99].

In this dissertation, the RTP model is chosen for simulations. The cost of power from the grid at any time, t, is the product of the instantaneous grid tariff and the power being drawn from the grid at that time interval multiplied by the time interval, Δt:

$$C_{P_g}(t) = G_t(t)\, P_g(t)\Delta t \tag{4.9}$$

It is important to note that the utility policy may forbid grid-connected battery energy storage systems from being charged from the distribution grid. It is assumed in this study that such a policy is not imposed. The RTP model and the degradation cost consideration already imposes cost constraints. The RTP tariff system increases the tariff during high load demand and reduces it

when the load demand goes down. Consequently, it is uneconomical for the system to recommend charging of the battery using the power from the grid when the tariff it high.

b) **Cost of degradation of the stationary battery storage system**

Despite their vital role in renewable energy deployment, battery energy storage systems are very expensive. Ignoring the cost of degradation when modeling the operational cost of the charging station would result in a costly charging system operation [17]. Studies show that the battery degradation cost is directly proportional to the fractional cycle life decline caused by the charge and discharge of a battery [100]:

$$C_{bd} = C_{bt}\frac{\Delta L_t}{L_t} \tag{4.10}$$

in which C_{bt} is the battery's initial cost, L_t is the battery total cycle life and ΔL_t is the change in cycle life due to the charge/discharge profile under investigation.

The stress factors affecting battery life are temperature, charge or discharge rate (power), charging time, time taken at low or high SoC, current ripple, depth of discharge (DoD) and average state of charge [5]. This study considers three main stress factors, namely, average SoC, DoD and temperature, and their contributions to the battery degradation cost.

- **Impact of Ambient Temperature on Battery Life**

As power is supplied to and drawn from the battery, the corresponding temperature fluctuations have deleterious effects on the health of the battery [101]. Badawy and Sozer [17] derived a simple thermal model of a battery based on the ambient temperature of the battery pack. The thermal model gave the battery operating temperature, T, as a function of the ambient temperature of the battery pack, T_a, the thermal resistance of the battery pack, R_t and the absolute average power during the charge and discharge cycles, $P_{bss}(t)$.

$$T = T_a + R_t|P_{bss}(t)| \tag{4.11}$$

The *Arrhenius* equation suggests an exponential relationship between the battery life to its capacity and temperature according to [102]:

$$L_t(T) = \alpha Q exp\left\{\frac{\beta Q}{T}\right\} \tag{4.12}$$

where $L_t(T)$ is the battery lifetime, Q is the battery rated capacity, T is the operating temperature and α and β are curve fitting constants. Consequently, the study in [17] defined the temperature contribution to the cost of battery degradation, C_T as:

$$C_T = C_{bt} \int_{t_o}^{t_f} \frac{dt}{Y_h L_t(T)} \tag{4.13}$$

where C_{bt} is the initial capital investment on the battery, Y_h is the number of hours per year and t_o and t_f are the initial and final times of use of the battery. The battery lifetime-temperature dynamics used in this dissertation are borrowed from [17].

- **Effect of Depth of Discharge on Battery Life**

Battery stress due to depth of discharge (DoD) is one of the greatest contributors to battery degradation [103]. The DoD of a battery is defined as the ratio of energy drawn from the battery to the rated capacity. The instantaneous DoD is therefore expressed as a function of the battery SoC according to the equation below [8]:

$$DoD(t) = 1 - SoC(t) \tag{4.14}$$

If this equation were used as it is, then the optimization episode will begin with an initial DoD which is not a result of the charge and discharge operations recommended by the controller, making the resultant cost erroneous. *Xu* [103] defined the depth of discharge as the absolute fractional change in the state of energy (SoE). The SoE is the amount of energy available in the battery. It may be computed by $SoE(t) = E_b SoC(t)$. Then, the depth of discharge is computed as follows [103].

$$DoD(t) = \frac{|SoE(t) - SoE(t-1)|}{E_b} \tag{4.15}$$

To define the cost of degradation due to DoD-related stress on the battery, we need to relate it to the battery's cycle life decline. Zhou *et al.* [104] established that for a Lithium-ion battery, the number of life cycles grows exponentially with a decrease in DoD as given by [104]:

$$L(DoD) = \alpha DoD^{-\mu} \tag{4.16}$$

A study of the Lithium-ion battery data established that the values of α and μ could be 694 and 0.795, correspondingly [104]. According to Abronzini *et al.* [8], the cost of cycle life degradation as a result of operating a battery from DoD_1 to DoD_2, for $DoD_2 > DoD_1$, can be approximated as follows:

$$C_{DoD} = \left(\frac{1}{L(DoD_2)} - \frac{1}{L(DoD_1)}\right) C_{bt} \tag{4.17}$$

$L(DoD_j)$ being the expected cycle life of a battery operated at DoD_j. Thus, short charge/discharge cycles are cheaper than long ones.

- **Effect of Average State of Charge on Battery Life**

The average SoC of the stationary battery for a time step from *t* to *t+1* is given by:

$$SoC_{av} = \frac{SoC(t) + SoC(t+1)}{2} \tag{4.18}$$

Hoke et al. [100] considered a 15-year data on battery capacity against average SoC data and came up with a model of costing the capacity fade degradation. The authors reported that the cost of capacity fade of the battery due to constant charge and discharge cycles is proportional to the average SoC and is given by:

$$C_{SoC} = C_{bt} \frac{mSoC_{av} - d}{Q_{fade} n Y_h} \tag{4.19}$$

where Q_{fade} is the capacity fade, SoC_{av} is the average SoC and m, n and d are curve fitting constants.

Therefore, the cost of battery degradation is taken as the maximum value of the cost contributions of each of the considered stress factors [17]. Curve fitting details are available in [17].

$$C_{bd} = max\{C_T, C_{DoD}, C_{SoC}\} \tag{4.20}$$

Or

$$C_{bd}(t) = C_{bt} max\left\{\left(\int_{t_o}^{t_f} \frac{dt}{Y_h L_t(T)}\right), \left(\left[\left(\frac{1}{L(DoD_2)}\right) - \left(\frac{1}{L(DoD_1)}\right)\right]\right), \left(\frac{mSoC_{av} - d}{Q_{fade} n Y_h}\right)\right\} \tag{4.21}$$

And the cost of degradation of the BSS at any timestep is:

$$C_{P_{bss}}(t) = P_{bss}(t)C_{bd}(t)\Delta t \quad (4.22)$$

It is reasonable to take the maximum of any of the three because the costs are highly interdependent and independently factoring them in the degradation cost equation is quite a challenging task.

4.3 Grid-tied Microgrid Power Scheduling as a Markov Decision Process (MDP)

For the optimal scheduling task to be achieved by an RL algorithm, it is first expressed as an MDP. The states need to have a Markov property so that the current state has enough information to make a decision [47], [54]. An MDP is defined by its state space, action space, reward model and the state transition model. For the problem in this study, the MDP components are defined as follows.

i) State and State Space

The system state, x_k defined as:

$$x_k = \{P_{cl}^k, P_{pv}^k, G_t^k, E_b^k\} \quad (4.23)$$

where k is timestep index such that $k = 0,1, \dots, T-1$. K is also a state index; thus, T denotes the total number of individual states in a single episode. P_{cl}^k, P_{pv}^k, G_t^k and E_b^k are the load at the station, PV power, the unit cost of energy from the utility and the amount of energy in the BSS at time k respectively. The state-space is thus defined as a union of all the set of states in T: $\chi = x_0 \cup x_1 \cup , \dots, \cup x_{T-1}$ [21]

ii) Action and Action Space

The RL agent is modeled to decide on the power schedule at the charging station given any state in an episode. Conventionally, battery scheduling using the Q-learning approach defines action space in two major ways.

i) The battery can be modelled to only have three possible control actions, namely, charge, discharge and idle [45], [105]. If the recommended action is a charge or a discharge operation, the battery charges or discharges at full rate, otherwise if the endorsed action is to idle, then the battery power is zero. This reduces the action space considerably and makes the learning process simple and efficient. However, the results returned may be sub-optimal if the choice of charging rate is not properly done. It is most applicable for the

backing storage (main storage) in a two-level storage system as demonstrated in the studies in [45] and [49]. Two-level storage systems consist of a main storage and an auxiliary storage [49].

ii) The battery energy levels (state of energy) or state of charge may be discretized from a minimum value to a maximum value in defined steps [17]. For instance, a 100 kWh battery may have 9 possible levels from a minimum value of 10 kWh in steps of 10 kWh. An arbitrary choice of the initial battery energy is made, then the next values of the battery energy are selected by the learning agent at every time step. The choice of the next state of energy of the BSS is the determinant of the amount of power to be supplied to or drawn from the it during the time step to ensure the battery reaches the selected level is calculated and scheduled for dispatch.

This study further proposes an action-space model that is not fixed but varies from state to state depending on the power requirements and the deficit that needs to be purchased from the grid or excess power to be supplied to the grid. The objective is to improve the convergence and stability of the Q-learning algorithm. We define the action as a two-tuple, that is,

$$a_k = \{P_{bss}^k, P_g^k\}. \tag{4.24}$$

where P_{bss}^k and P_g^k denote the battery and the grid power respectively. Therefore, the action space for every state depends on the state variables: $\mathcal{A}_k = f(x_k)$. The action space is limited by the grid power and battery energy level according to: $P_g^{min} \leq P_g^k \leq P_g^{max}$ as well as $E_b^{min} \leq E_b^k - P_{bss}^k \Delta t \leq E_b^{max}$.

To delimit the action space in every state, considering the load and the PV power available, the power deficit is calculated by:

$$\Delta P_k = P_{cl}(k) - P_{pv}(k) \tag{4.25}$$

Given the value of the power discrepancy above, the algorithm computes all possible combinations of P_{bss}^k *and* P_g^k such that:

$$\Delta P_k = P_{bss}^k + P_g^k. \tag{4.26}$$

This is done using a simple algorithm given in Fig. 4.2. In the pseudo-code, P_g^{min}, P_g^{max}, $P_{bss}^{k_min}$ *and* $P_{bss}^{k_max}$ are the lower and upper bounds of grid power and battery power respectively, and

step_g and *step_b* are the discretization steps for grid and battery power respectively. If ΔP_k is negative, then there is excess PV generated, thus the battery or the grid or both will have negative power to maintain the power balance at the DC link. Otherwise, if ΔP_k is positive, then there is insufficient PV power, therefore, the battery or grid power will have to be positive to sustain the power balance at the DC bus.

The restriction of the list of possible control actions per given state helps the agent to avoid trying actions that are not viable within the constraints of the charging station. This enhances the stability of the learning process and hastens the convergence of the algorithm. The action space for the entire environment is defined by $\mathcal{A} = \mathcal{A}_0 \cup \mathcal{A}_1 \cup, \ldots, \cup \mathcal{A}_{T-1}$.

1. *Define an empty possible actions list*
2. *Define* P_g^{min}, P_g^{max}, $P_{bss}^{k_min}$ *and* $P_{bss}^{k_max}$
3. *for* P_g^k *in range* (P_g^{min}, P_g^{max}, *step_g*):
4. *for* P_{bss}^k *in range* ($P_{bss}^{k_min}$, $P_{bss}^{k_max}$, *step_b*):
5. *if* $P_{bss}^k + P_g^k = \Delta P_k$:
6. *Add vector* [P_{bss}^k, P_g^k] *to the list*
7. *end if*
8. *end for*
9. *end for*
10. *End*

Fig. 4.2: Action space determination algorithm

iii) State Transition Function

The state transition function determines the next state, given the current state and the action. For an MDP, the next state depends only on the present condition of the environment and not a history of events that produced the current state [46]. The state transition in this study is modeled to be deterministic, i.e., $x_{k+1} = f(x_k, a_k)$ [11]. x_{k+1} denotes the vector containing the system inputs for the next state, i.e., the load P_{cl}^{k+1}, the output of the PV generator, P_{pv}^{k+1}, and the utility tariff G_t^{k+1} as well as the BSS energy level adjusted according to the action. The next value of the BSS energy is given by:

$$E_b^{k+1} = E_b^k \pm P_{bss}^k \Delta t. \tag{4.27}$$

And the next state x_{k+1} is given by:

$$x_{k+1} = \{k+1, P_{cl}^{k+1}, P_{pv}^{k+1}, G_t^{k+1}, E_b^{k+1}\} \tag{4.28}$$

iv) Reward Function

A reward helps to communicate the objective of the learning process to the agent. Intelligent reward development for the achievement of the objective of the RL algorithm [82]. The reward is defined as $r(k) = g(x_k, a_k, x_{k+1})$.

The learning process intends to minimize the operational cost of charging EVs, i.e., minimizing the cost of purchasing power from the utility while maintaining the health of the BSS. Intuitively, the reward can be expressed as negative of the cost [34] as given in Equation (4.29).

$$r(t) = -(C_{P_g}(t) + C_{P_{bss}}(t)) \tag{4.29}$$

While that kind of reward works well for deep reinforcement learning methods, it does not produce a good learning characteristic with a Q-table based method. This is because the Q-table is initialized with zeros. Updating the Q-values using negative rewards causes the Q-values for actions that have been tried to be negative. Consequently, the actions that have not been tried have high Q-values, which forces the agent to explore them even when the action-selection algorithm has recommended exploitation. Thus, the reward function for the Q-learning algorithm is given as:

$$r(t) = \frac{1}{\left(C_{P_g}(t) + C_{P_{bss}}(t) + 1\right)^2} \tag{4.30}$$

In Equation (4.30), the reward is expressed as the inverse of the square of the cost. When the agent obtains a policy that maximizes this reward, the operational cost is minimized as a consequence of the agent's actions. The addition of 1 in the denominator helps to avoid division by zero which may occur when there is no charge or discharge power scheduled and no power is drawn from or supplied to the grid. The behavior of the two reward functions (Equation (4.29) and Equation (4.30)) are explored in this dissertation.

4.4 Conventional Q-learning Solution to the Power Scheduling Problem

Conventionally, a Q-table is used to track the learning of the agent in Q-learning. The Q-table is a matrix with rows equal to the number of states and columns equal to the number of possible actions. Both the state and the action spaces are stationary. Therefore, the agent visits the states successively and synchronously. In each time step, it picks an action from the list of allowable actions in each state, the Q-value for the pair of state and action is updated. Table 4.1 gives an illustration of a classical Q-table.

Table 4.1: Q-table for the conventional Q-learning method

State/Action	**a_1**	**a_2**	**a_n**
s_1	$Q(s_1, a_1)$	$Q(s_1, a_2)$	$Q(s_1, a_n)$
s_2	$Q(s_2, a_1)$	$Q(s_2, a_2)$	$Q(s_2, a_n)$
S_n	$Q(s_n, a_1)$	$Q(s_n, a_2)$	$Q(s_n, a_n)$

The conventional Q-learning algorithm to solve the power management problem is described in the algorithm in Fig. 4.3.

For this power scheduling problem, the actions are the possible discrete energy levels of the BSS from 10.0 kWh to 100.0 kWh in steps of 10.0 kWh. The value of the BSS energy is initialized arbitrarily. In each state, the agent chooses the next value of battery energy level. Given this value, the battery power during the given time step may be calculated to drive the battery to the energy level selected by the agent. Then the power equilibrium Equation (2.1) is applied to calculate the amount of power to be drawn from or sent to the grid. The cost of degradation of the BSS as well as the cost of power from the grid are then computed. The agent then transits to the next state. The energy cost is used to compute the reward according to Equation (4.30). The Q-value of the selected action is adjusted as given in Equation (3.18) or (3.19) contingent on the current state being the end of an episode or not. At the end of every episode, the value of the exploration rate, ϵ, is updated according to the ϵ-greedy algorithm.

1. *Initialize learning parameters* (α *and* γ*)*
2. *Initialize epsilon* ($\epsilon = 1$)
3. *Initialize q-table (Q= 0 for all actions)*
4. *For n = 0 to maximum iteration:*
5. *k = 0*
6. *Get the initial state vector*
7. *For k = 0 to T-1:*
8. *Take action by* ϵ *−greedy method*
9. *Find next state (using state transition function)*
10. *Calculate reward (using reward function)*
11. *Update q-value for state-action pair*
12. *k = k + 1*
13. *End for*
14. *Update* ϵ
15. *n = n + 1*
16. *End for*
17. *Return q-table*

Fig. 4.3: Conventional Q-learning for power scheduling

The ϵ-greedy method is considered for purposes of simplicity. The choice of the number of episodes is done by trial and error until the algorithm converges to a maximum reward, and by extension, a minimum cost.

According to the ϵ-greedy action selection strategy, the value of ϵ is set at a value as close as possible to 1 and decreased slowly at the end of each episode. In this study, an exponential decay function is used to reduce ϵ as given by Equation (4.31).

$$\epsilon = min_\epsilon - (max_\epsilon - min_\epsilon)exp\{-\mathbb{C} \times n\} \quad (4.31)$$

In the equation, min_ϵ and max_ϵ are the minimum and the maximum values of ϵ respectively, while $\mathbb{C}$ is the cooling constant and n is the episode number.

4.5 Proposed Asynchronous Q-learning Solution for the Power Scheduling Problem

An asynchronous Q-learning is proposed to solve the power scheduling problem. Instead of the classical Q-table with a fixed number of rows and columns, a hash table that grows with time is developed. The hash table is initialized as an empty Python dictionary data structure, i.e., a structure in the form of keys and their corresponding values. In the Q-dictionary, states are added as keys. The value for each key is another dictionary, i.e., the dictionary of possible actions. For each dictionary of possible actions, an action is a key and the Q-value is the value. The Q-table, therefore, becomes a nested dictionary, with states indices as the main keys and an inner dictionary of possible actions as values. Each action has its Q-value initialized with zero. States are added as they occur during learning, thus, the agent avoids states that are never visited by the agent.

At the beginning of the learning, the hash table is empty. The algorithm gets the initial state. If the state had not been in the hash table, the algorithm finds the allowable actions in that state and creates a dictionary containing the actions as keys and Q-values, initialized with zeros, as values. The ϵ-greedy algorithm is used to pick and action. The next state is then found using the state transition function. The reward is then computed. The Q-value for the pair of state and action is adjusted using Equation (3.18) or (3.19) depending on whether or not the state is a final state in the current episode. The algorithm only includes the states that have been visited by the agent in the Q-dictionary. The learning process is asynchronous because of the fact that the agent does not sweep through the whole state space in very iteration and the states are accessed quasi-sequentially. The algorithm proceeds as shown in Fig. 4.4.

At the end of the learning process, the algorithm returns a Q-table which now contains states, possible actions and their corresponding Q-values. The process of getting the optimal action for each state using this Q-table is called policy retrieval. It is accomplished by iterating through the states in the Q-table from $k = 0$ to T-1. In every state, the control action that maximizes the Q-value is retrieved. Then the following states are the ones to which the selected control action leads. The states with their corresponding optimal schedules of battery power and grid power are returned. This policy retrieval process is illustrated in Fig. 4.5.

1. *Initialize learning parameters* (α *and* γ)
2. *Initialize epsilon* ($\epsilon = 1$)
3. *Create an empty q-table*
4. *For n = 0 to maximum iteration:*
5. *k = 0*
6. *Get the initial state vector*
7. *For k = 0 to T-1:*
8. *If the state is not in q-table:*
9. *Get possible actions*
10. *Set Q = 0 for all actions in the state*
11. *Add state-possible actions pair to q-table*
12. *End if*
13. *Take action by* $\epsilon -$*greedy method*
14. *Find next state*
15. *Calculate reward*
16. *Update q-value for state-action pair*
17. *k = k + 1*
18. *End for*
19. *Update* ϵ
20. *n = n + 1*
21. *End for*
22. *Return q-table*
23. *End*

Fig. 4.4: Asynchronous Q-learning for power scheduling

1. *Create an empty list of states and actions*
2. *Read the initial state*
3. *k = 0*
4. *For k = 0 to T-1:*
5. *Add state to list of states*
6. *Get greedy action*
7. *Add action to list of actions*
8. *k = k + 1*
9. *Read the next state*
10. *End for*
11. *Return list of states and corresponding greedy actions*

Fig. 4.5: Policy retrieval algorithm

4.6 Summary

The power scheduling problem has been formulated as a mathematical optimization problem in this chapter. A comprehensive battery degradation model has been provided. Also, the charging station's operation has been modeled as a Markov Decision Process. An asynchronous Q-learning has been proposed to obtain optimal power schedules. The conventional Q-learning approach has also been described for purposes of comparison. The algorithms developed in this chapter employ Q-tables to store Q-values for state-action pairs. Although the Q-table method of learning is simple and has a guarantee of convergence, it is unscalable. Furthermore, the performance of the algorithm depends on the action-selection strategy employed. In the next chapter, a deep reinforcement learning algorithm is explored. The objective is to improve the scheduling results in terms of cost and scalability and learning characteristics using an advantage actor-critic (A2C) algorithm.

5.0 Power Scheduling Using Advantage Actor-Critic Algorithm

In this chapter, the application of an advantage actor-critic algorithm in solving the optimal power scheduling problem is explored. The main objective of this chapter is to improve the results of the scheduling task with respect to the convergence to the global cost using deep reinforcement learning techniques. Modern techniques of improving learning in deep reinforcement learning such as experience replay and parallel learning are explored.

5.1 Background

Reinforcement learning algorithms make use of a scalar reward signal to enable a learning agent to acquire knowledge of a policy that maps the states of their environment into optimal actions. In Q-learning, a value function is used to learn the optimal policy, such that the higher the value of an action in a state, the higher the probability of selecting that action. The method is efficient and has guaranteed convergence. However, a Q-learning method that employs a lookup table to track learning progress has limitations of lack of scalability and inability to operate online [18]. Furthermore, the action-selection strategies used to balance exploration and exploitation may affect the accuracy of the final results [44].

The application of function approximation methods using artificial neural networks (ANNs) has been proposed to address the above-mentioned drawbacks. However, applying linear function estimation techniques to the state-action value techniques has been shown to display unsatisfactory convergence and in the case of off-policy methods such as Q-learning, it may result in a divergence [60]. The other issue with the value function-based algorithms is that the dilemma of exploration and exploitation remains a major challenge. Furthermore, value-based methods such as deep Q-network (DQN) and Q-learning show slow convergence [18].

Policy gradient algorithms use ANNs to approximate the policy directly. The parameters of the ANNs are adjusted in the direction that recommends actions that maximize the total returns. Thus, the policy gradient algorithms have been reported to converge faster than value-based methods [91]. Traditionally, policy gradient methods required that a policy is obeyed all through the episode before an update can be executed as it is done in Monte Carlo learning [60]. However, waiting for an episode to end before performing an update may produce inaccurate results due to high variance

in the policy approximations. An episode may return high rewards but with a few bad (even risky) actions that may have even returned negative rewards. Due to stochasticity in the policy being obeyed in each episode, a variance prone learning method like the Monte Carlo technique may fail to converge to the true global optimum [50].

Actor-critic algorithms combine value-based methods with policy gradients. Instead of waiting for an episode to be concluded, the algorithm performs updates in each time step [44]. The actor-critic architecture allows for a feedback mechanism that helps with seamless online error correction. The combination of the power of ANNs and online error correction is such an attractive tool in microgrid power scheduling in dynamic environments. Lately, actor-critic methods have gained popularity in microgrid power scheduling due to their efficiency and capacity to operate online. In [73], an AC algorithm is employed to optimize power allocation in an assorted power network with wind turbines and PVs, leading to a significant increase in energy efficiency. Wan *et al.,* [74] scheduled energy storage in a smart home using an AC algorithm leading to a significant energy cost reduction. However, conventional AC methods are still prone to being trapped in a local optimum due to high bias in policy updates, thus may converge to a lower global cost [92].

In this chapter, an advantage actor-critic (A2C) algorithm is applied in solving the scheduling task under static conditions. In this algorithm, modern deep reinforcement learning techniques such as experience replay and parallel learning are employed to improve convergence. It is shown that introducing experience replay and parallel learning techniques in A2C produces better convergence than the conventional actor-critic algorithm. This chapter builds on the studies in [106] and [107] that used experience replay and parallelization of learning agents to improve learning outcomes in the actor-critic algorithms.

5.2 Conventional Actor-critic Algorithm

The actor-critic architecture combines value function methods with policy gradients. This hybridization enables the algorithm to borrow the advantages of both methods and minimize their limitations. The algorithm consists of two neural networks, namely, the actor and the critic. The actor uses the policy gradient theorem to learn a policy and the critic applies the Bellman update method to approximate the value function.

The actor takes the environment's state as input and produces a probability distribution (policy), $\pi(\omega)$, by which an action is selected. The critic takes the same state and estimates the value

function $V(x_k, \theta_n)$. The action selected is executed and the next state is generated from the environment and the reward $r(k)$ is computed. The next state is forwarded to the critic network to get the value of the next state $V(x_{k+1}, \theta_n)$. TD error $\partial(x_k)$ is then computed using equation (3.36) and used to update the critic network.

If θ represents the parameters of the critic network and α its learning rate, then it is updated using stochastic gradient descent, and in the direction that minimizes the TD error as follows.

$$\theta_{n+1} = \theta_n + \alpha\partial(x_k)\,\nabla_{\theta_n} V(x_k, \theta_n) \tag{5.1}$$

The actor network is updated using the policy gradient method. If ω represents the weights of the actor network and β its learning rate, then it is updated using stochastic gradient ascent and in the direction that maximizes the total expected discounted rewards that as follows:

$$\omega_{n+1} = \omega_n + \beta\partial(x_k)\nabla_{\omega_n} log\,(\pi(\omega_n)V(x_k, \theta_n)) \tag{5.2}$$

The loss in the value estimation (critic loss) is the square of the TD error, i.e., the square of the difference between the predicted state value, $V(x_k, \theta_n)$, and the target state value, $R(k) = r(k) + \gamma R(k+1)$. The loss in policy estimation (actor loss) is given by:

$$L_w = -\partial(x_k) log\,(\pi(\omega_n)) \tag{5.3}$$

The conventional AC algorithm is given in Fig. 5.1 [93].

5.3 Advantage Actor-critic Algorithm

Compared to the conventional AC algorithm, the A2C algorithm is the same in overall architecture but different in update procedure. In the case of an A2C algorithm, $V(k, \theta_n)$, is substituted by the advantage function given by:

$$A(x_k, a_k) = Q(x_k, a_k, \theta_n) - V(x_k, \theta_n) \tag{5.4}$$

To apply the experience replay in the actor-critic framework, the agent interacts with the environment and gets transition tuples for a given number of steps or until the end of an episode. After these steps, the mean values of the policy losses and value losses are calculated and used to update the agent's actor and critic networks. The value loss is the mean squared TD error. The policy loss is computed using the advantage function as:

$$L_w = \frac{1}{T}\sum_k [-A(x_k, a_k, \theta_n)log\ (\pi(\omega_n))] \tag{5.5}$$

1. *Initialize actor and critic network weights*
2. *Set actor and critic learning rates*
3. *For episode = 0 to maximum iterations*
4. *Get initial state*
5. *For k = 0 to end of episode (T-1):*
6. *Take action using the current policy*
7. *Get the next state*
8. *Compute the reward*
9. *Calculate the return ($R(k) = r(k) + \gamma R(k+1)$)*
10. *Calculate the TD error*
11. *Calculate the actor (policy) and the critic (value) losses*
12. *Update actor and critic networks*
13. *End for*
14. *End for*
15. *End*

Fig. 5.1: Conventional Actor-Critic Algorithm [93]

Policy gradients may easily be ensnared in a local extremum and consequently, the learner may cease to explore, causing the policy to not change. This may lead to sub-optimal results. The solution to this is by adding an entropy bonus to the actor loss. Entropy is the term used to describe the degree of uncertainty in the probability distribution returned by the actor network. The entropy bonus is computed as shown in Equation (5.6).

$$\varepsilon(P) = \sum_i p_i log(p_i) \tag{5.6}$$

where the upper case P represents the probability distribution, p_i is the value of each element in the distribution. This adds some noise to the actor loss thus encouraging the actor to explore the action space, thus preventing the policy from getting to a biased deterministic rule. The A2C algorithm is shown in Fig. 5.2.

1. *Begin*
2. *Initialize actor and critic network weights*
3. *Set actor and critic learning rates*
4. *Set update steps to value (D)*
5. *For episode = 0 to maximum iterations*
6. *Create an empty replay buffer*
7. *Get initial state*
8. *For k = 0 to end of episode (T-1):*
9. *Take action using current policy*
10. *Get the next state*
11. *Compute the reward*
12. *Store the set (state, action, next state, reward) in the replay buffer*
13. *If k = T-1 or k = D:*
14. *For each experience set (starting from the last backwards)*
15. *Estimate values of each current and next state*
16. *Calculate the return (* $R(k) = r(k) + \gamma R(k+1)$ *)*
17. *End for*
18. *Compute the advantage for each state-action pair*
19. *Calculate the actor (policy) and the critic (value) loss*
20. *Update actor and the critic using backpropagation*
21. *End if*
22. *End for*
23. *End for*
24. *End*

Fig. 5.2: Advantage actor-critic algorithm

Line 14 in Fig. 5.2 means that the actor and the critic are updated using the losses returned by values obtained in reverse order in which states occurred at the end of an episode. This update may also be done at the end of a pre-determined step size, *D,* even before the end of an episode. The value of D determines the number of state transitions or experiences that are to be gathered before an update is done. It helps to strike a balance between pure Monte Carlo learning (that waits for

an episode to end) which has high variance and pure temporal difference update (as in conventional AC) which has high bias. Therefore, it helps in solving a bias-variance dilemma in the policy estimation.

In the A2C algorithm, the policy is obeyed in the forward direction. Then the updates are done as though they were experienced in the reverse order. Thus, the learning process mimics the case of workers interacting with different environments as in the A3C algorithm [106]. However, unlike the A3C algorithm, A2C parallelizes the learning without introducing the complexities of asynchrony.

5.4 Power Scheduling Using Advantage Actor-Critic Algorithm

5.4.1 Environment design

The first step in the application of a reinforcement learning algorithm is to develop an environment. The power management problem was defined and its MDP developed in the previous chapter. The environment used in this case will be the same as the one developed before. The only modification is that the time component is removed from the state variables since the value was only used as a state identity in the Q-table which is not needed in this case. Thus, the state is given by:

$$x_k = \{P_{cl}^k, P_{pv}^k, G_t^k, E_b^k\} \tag{5.7}$$

The action space is the list of all possible battery energy levels from 10 kWh to 100 kWh in steps of 10 kWh. An action in this algorithm is the next value of the battery energy level, E_b^{k+1}. A state transition occurs according to the next values of state variables in Equation (5.7) for state k+1 read from the forecasted data. The next state therefore has the values $x_{k+1} = \{P_{cl}^{k+1}, P_{pv}^{k+1}, G_t^{k+1}, E_b^{k+1}\}$. The action and the current state variable, E_b^k, is used to compute the battery power as follows:

$$P_{bss}(k) = \frac{E_b^{k+1} - E_b^k}{\Delta t} \tag{5.8}$$

Then, the value of the grid power is calculated using the power balance equation as follows:

$$P_g(k) = P_{cl}^k - P_{bss}(k) - P_{pv}^k \tag{5.9}$$

The cost of buying power from the utility and the battery degradation are then computed as previously given.

The objective of the learning process is communicated to the agent through a reward signal. In this case, the objective is to minimize the energy cost in charging an electric vehicle. Normally, the reward is computed as negative of the cost in such situations. Reward function of this nature is a simpler and better representation of the learning objective and is popular with deep reinforcement learning methods [34], [74]. The reward in this case given by:

$$r(k) = -\left(C_{P_g}(t) + C_{P_{bss}}(t)\right) \tag{5.10}$$

5.4.2 **Model of the Actor and Critic Networks**

The actor and the critic networks consist of an input layer, a hidden layer and an output layer. Both the actor and critic networks take the state variables as input. Therefore, the input layer for both networks had an input node for each state variable, i.e., 4 input nodes for each neural network. There is a wide range of values of the number of nodes in the hidden layer that can produce good results in this problem. This is because reinforcement learning is not concerned with the absolute values of the neural network predictions, but the indication of how better or worse the current policy or value is compared the previous policy or value. Different values of number of nodes were tried from 50 to 300 in steps of 50 (See Fig. A.1 in Appendix A). Values between 150 and 300 displayed similar learning curves except for small differences in the trajectory due stochasticity in the policy. Values below 100 converged after slightly more number of episodes but still returned a global costs that were not significantly different from those obtained from higher number of nodes. In the design of the A3C algorithm from which A2C was borrowed, a value of 256 has been used to solve different problems [106], [108]. Thus, a value of 256 was chosen to perform simulations in this dissertation. The output layer contains as many output nodes as the number of actions in the action space, i.e., each node outputs the probability of taking one of the 9 possible actions (action space, $A = 9$). Fig. 5.3 and Fig. 5.4 show the actor and the critic models respectively.

Although the two neural networks take the same inputs, their updates processes are totally different and that informs how their weights adapt to produce different outputs. The critic network is updated using the stochastic gradient descent method as given in Equation (5.1) while the actor network is updated using the stochastic gradient ascent method as given in Equation (5.2). For each forward pass in the each of network models, the ReLU activation function is used at the input

through to the hidden layer. However, for the output layer, the data is transformed using the SoftMax function for the actor to return a probability distribution that adds up to unity while that for the critic is output as it is without activation.

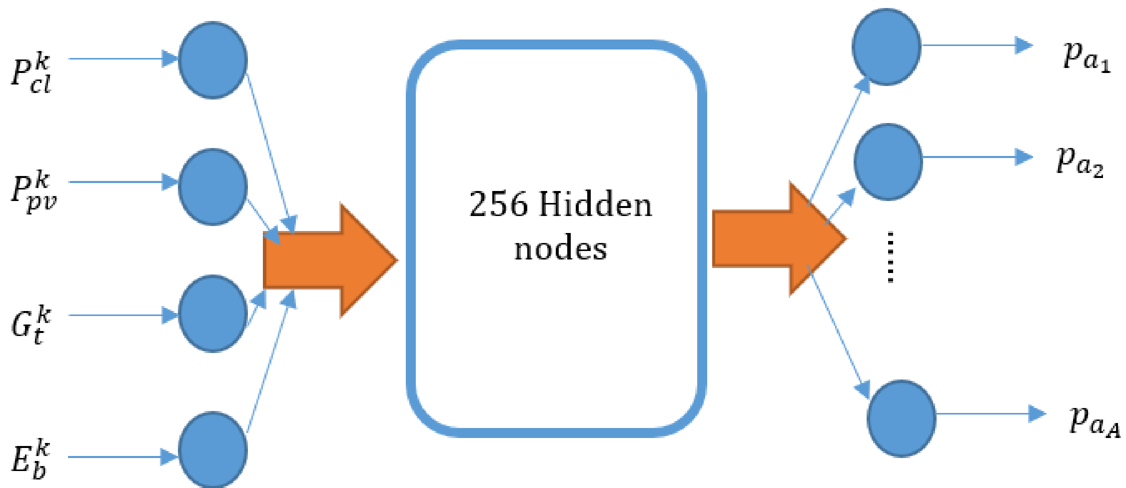

Fig. 5.3: A2C actor model

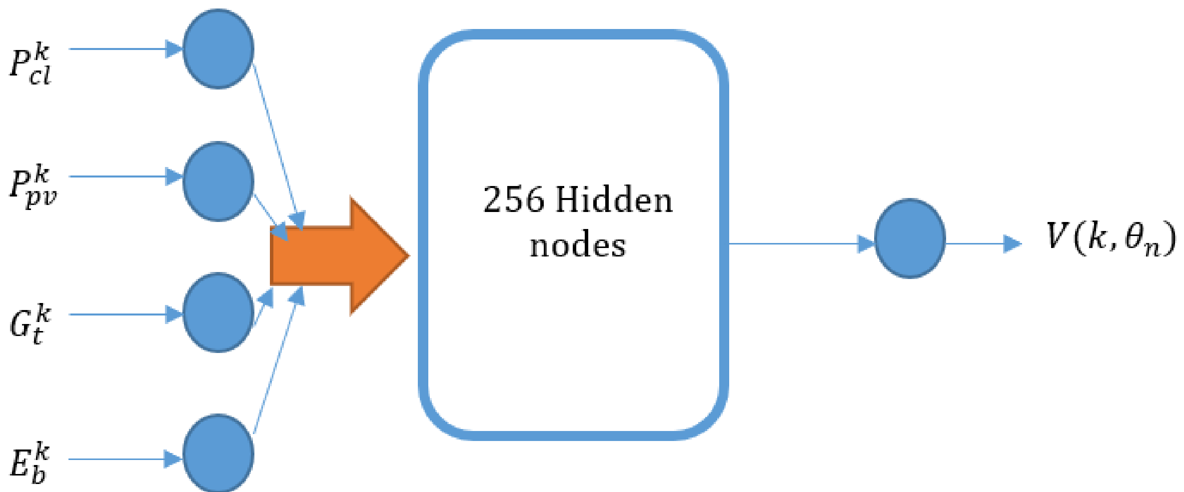

Fig. 5.4: A2C critic model

5.4.3 **Training of A2C**

The neural networks in actor-critic techniques are trained using the back-propagation algorithm. The algorithm calculates the gradient of a loss function with respect to each element in a neural network's weight vector. The objective of backpropagation is to minimize the loss starting from

the output layer backwards [89]. The learning process begins by initialization of the actor and critic weights with arbitrary values. The agent's learning rates and the update step (D) are initialized. For each episode, an empty replay buffer is created. The initial state is read, and the value of battery energy is initialized (initial value is 50 kWh, selected arbitrarily and for uniformity). For every state in an episode, the state is forwarded to the actor network and the current policy in the actor network is used to select the action, i.e., the probability distribution returned by the actor is used to pick an action from the list of possible actions. The list consists of battery energy values from 10 kWh to 100 kWh in steps of 10 kWh. The action determines the next value of battery energy from the current value. The state transition is then affected using this action. Also, using the action, the battery power and the grid power are computed. The reward is then calculated. The transition elements, i.e., the state, the action, the reward and the next state are then stored in the replay buffer. If the number of transitions is equal to the step size D or the episode length T-1, then starting from the last transition and working backwards, the return and state values are computed. Then TD error and the value of the advantage, $A\ (x_k,\ a_k)$, are computed for each transition. The mean square TD error is calculated to represent the loss in the value estimation (critic loss). The loss in the policy estimation (actor loss) is then computed using Equation (5.5). The actor and the critic networks are then updated using the respective losses through the backpropagation algorithm.

5.5 Summary

In this chapter, an advantage actor-critic algorithm has been developed to perform static power scheduling in a grid-connected PV-battery electric vehicle charging station. The chapter has explained the application of experience replay and parallelization of learning in improving the performance of the actor-critic architectures. Also, the conventional actor-critic method, which does not employ these techniques, has been described for comparison. In the next chapter, the developed algorithms are tested and evaluated. The simulation results are also analyzed and discussed.

6.0 Simulation Results and Analysis

In this chapter, the algorithms developed in Chapter 4 and Chapter 5 are tested using a typical PV-battery EV charging station's data. The computing platform and the test data used to perform simulations to evaluate the algorithms is presented. The results obtained through the developed algorithms are then compared with their conventional counter parts in terms of the learning characteristics, scalability and global cost. Also, the optimized power schedule obtained from the proposed algorithms are presented and analyzed. Finally, the results are discussed. The discussion highlights the implications of the results and the limitations of the algorithms.

6.1 Simulation Platform

All the algorithms were implemented on a 64-bit Windows 10 Intel (R) Core i7 CPU, 1.80GHz with 8GB RAM. Python programming language was used to implement the developed algorithms. The Python scripts were written and run on the Jupyter Notebook code editor in the Anaconda development environment. Python programming language was selected because it has specialized tools that make implementation of the machine learning algorithms easy and efficient. Some of the Python machine learning tools employed and their functions are in appendix C.

6.2 Input Data

6.2.1 Charging system parameters

Table 6.1 shows the system parameters of the charging station. The maximum and minimum grid power for each time step are dictated by a contract signed between the owners of the charging station and the utility. A negative value represents power supply to the grid and a positive value depicts power purchase from the grid. The choice of the timestep size is based on the data available for testing. The available data used in testing was in steps of 1hour. The Lithium-ion battery pack parameters for the purpose of simulations such as cost per kWh, temperature-lifetime dynamics, average SoC-cost characteristics and thermal resistance were taken from [17]. The initial battery energy (SoE_{in}) is normally selected arbitrarily. For this study, the value of SoE_{in} set at half the battery capacity.

Table 6.1: Charging station system parameters

Parameter	**Symbol**	**Values**
Minimum/Maximum grid power	P_g^{min}/P_g^{max}	-50/90 kW
Timestep size	Δt	1 hour
Battery cost per kWh	C_{bt}	$400
Battery capacity	E_b	100 kWh
Initial battery energy	SoE_{in}	50 kWh
Minimum battery energy	E_b^{min}	10 kWh
Battery thermal resistance	R_t	0.2 mΩ

6.2.2 **Test Data**

To test the developed learning algorithms, a typical PV-powered, EV charging station with a maximum PV output of 70 kW and a maximum load demand of 80 kW is considered as shown in Fig. 6.1 [21]. Level 3 DC charging, which is almost as fast as the filling process of an ICEV, requires a charging rate between 50 kW to 100 kW. Thus, the station is capable of limitedly supplying fast charging demands. The load, however, varies from time to time, with some vehicles demanding lower levels of charging rates such as level 1 (about 2 kW) and level 2 (8 kW to 20 kW) [24].

Depending on the immediate demand at the station, when the PV output is deficient to supply the demand, it is supplemented by the power purchased from the utility grid. If the PV harvest is in excess, the extra power is sent back to the utility grid. Also, as can be seen in Fig. 6.1, the grid tariff is dynamic. The dynamic tariff profile is an effective indicator of the load demand in the grid as it rises and falls with the load demand. Therefore, the grid tariff is highest during peak load and lowest during off-peak hours.

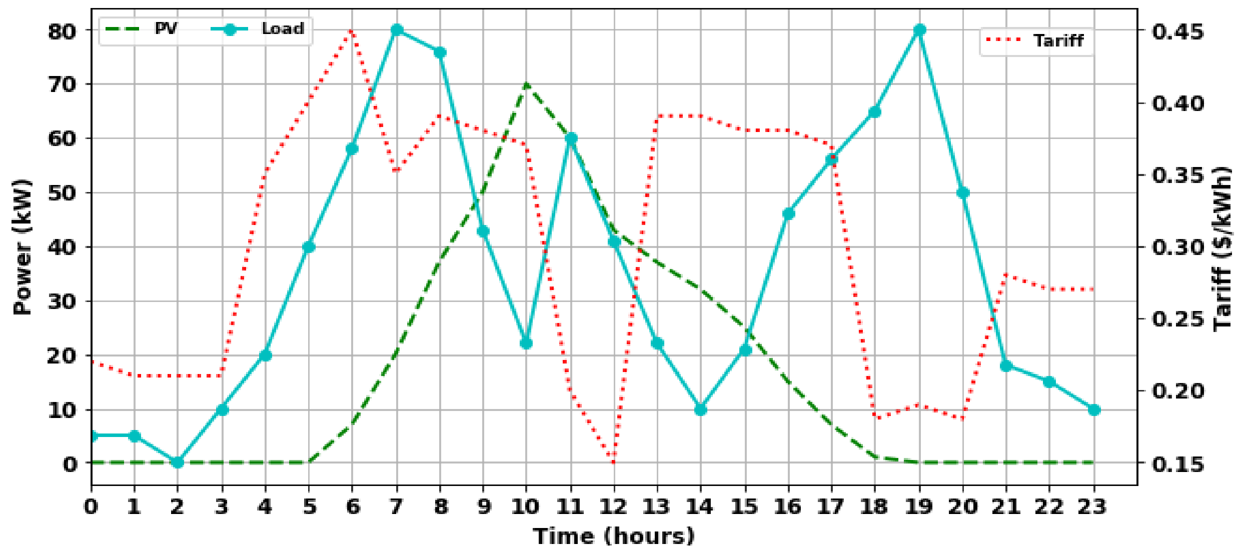

Fig. 6.1: Charging station's load, generation profile and the grid tariff profile

6.3 Simulation Results for Asynchronous Q-learning

6.3.1 Learning Parameters

The selected learning parameters are shown in Table 6.2. This sub-section explains how the parameters were chosen.

The exponential cooling of the value of epsilon helps to balance exploration with exploitation. Initializing epsilon with a value of 1.0 at the beginning of learning enables exploration of the search space since the probability of taking any of the actions is set to 1.0. The value of ϵ is then carefully reduced so that the actions which return the highest Q-values in every state have a higher probability of being selected. However, this value is not collapsed to zero because there is still a need to allow for some exploration in the later episodes just in case a better action than the current best is met by the agent. Thus, the minimum value of ϵ was set to 0.1. At this time, the greedy action has a probability of 0.9 of being chosen while the rest have a probability of 0.1 if being selected. The decay rate of ϵ is determined by the cooling constant, $\mathbb{C}$ which is chosen by trial and error to get the best convergence.

The learning rate, α, determines the rate at which the Q-values in the Q-table are modified during the learning iterations. To select the value of α, values from 0.0001 to 0.1 in multiples of 10 were

considered. A value of 0.001 was found to produce the best convergence. The effect of learning rate on the learning characteristics of the Q-learning algorithm has been discussed in Section 6.3.2.

Table 6.2: Learning parameters

Learning Parameter	Symbol	Chosen value
Initial exploration rate	max_ϵ	1.0
Minimum exploration rate	min_ϵ	0.1
Cooling constant	$\mathbb{C}$	0.00325
Learning rate (alpha)	α	0.001
Discount factor ($\boldsymbol{\gamma}$)	γ	1.0
Number of episodes	n	15000

6.3.2 Learning Characteristics of Q-learning

With the selected learning parameters and the charging station system constraints, the two Q-learning algorithms were trained. This section presents an analysis of the learning characteristics of the Q-learning algorithms during the training process.

RL algorithms are very sensitive to the learning rate (alpha). At each transition, the previous Q-value approximation is updated with the error between a new estimate and the previous guess. The value of alpha determines what percentage of this temporal difference error that is added to the previous Q-value owing to the agent's new experiences during learning. For the learning process to be stable and the convergence to be smooth, it is recommended that the value of alpha should be "sufficiently small" [11].

However, the choice of the value of the learning rate depends on the characteristics of the problem being solved such as the action space and the state space and the nature of the learning algorithm being employed. The stability of these Q-table based algorithms may be viewed from how sensitive they are to the learning rate.

Fig. 6.2 shows the learning characteristics of the conventional Q-learning algorithm with different values of alpha. The horizontal axis presents the moving average rewards calculated after every 100 episodes. The graph shows that for alpha = 0.1, the average reward is oscillating, and the agent does not acquire the intended policy to maximize the total rewards. Also, some instability can be observed for alpha = 0.01, though the convergence is much smoother. Lower values of alpha, i.e., 0.001 and 0.0001 display smooth convergence with the alpha value of 0.001 converging smoothly at higher total average reward than all the rest.

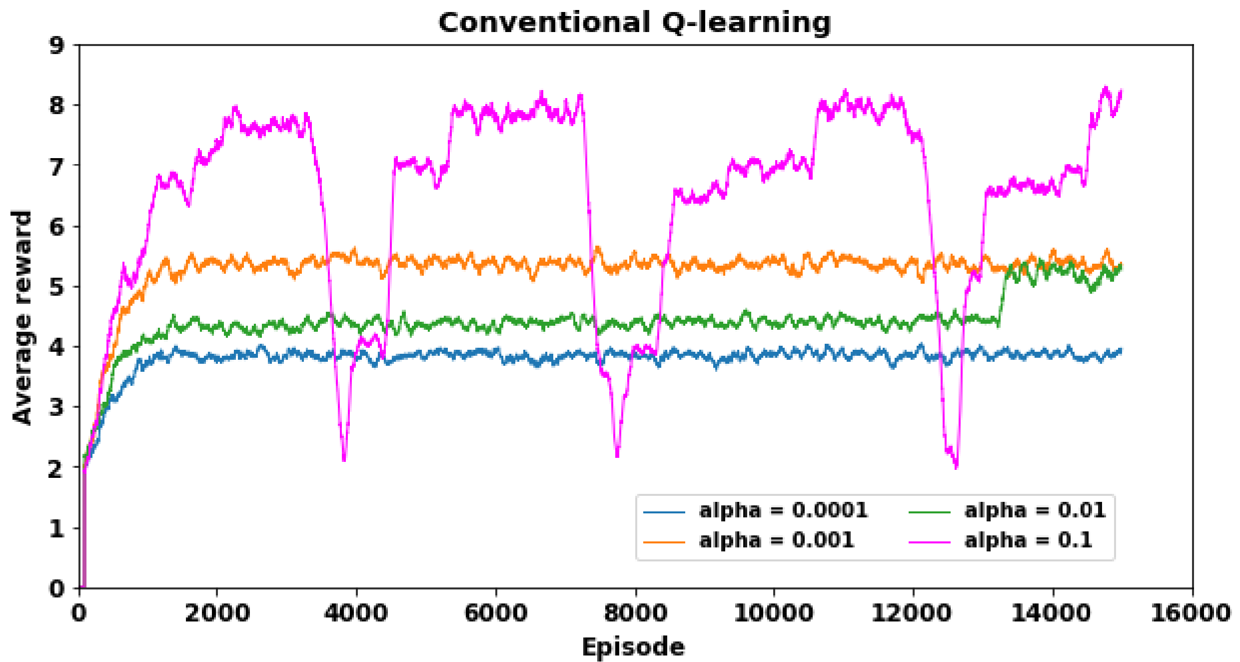

Fig. 6.2: Learning characteristics for the conventional Q-learning

Fig. 6.3 shows the learning characteristics of the asynchronous Q-learning algorithm for the same values of alpha as used above. An alpha value of 0.1 returned some oscillations on the average reward but the convergence was much more stable than that obtained by the conventional Q-learning. Fig. 6.3 reveals that the algorithm's learning curve for an alpha of 0.001 achieved the highest total rewards followed closely by an alpha of 0.0001. Therefore, for asynchronous Q-learning algorithm, a value of 0.001 was selected for the learning rate as in the conventional case.

Comparing the learning curves in Fig. 6.3 with those in Fig. 6.2, it can be seen that the curves for the various values of alpha were much closer together in Fig. 6.3 despite the big differences in the values of alpha. Therefore, the proposed asynchronous algorithm is seen to display a lower sensitivity to the learning rate than the conventional Q-learning method.

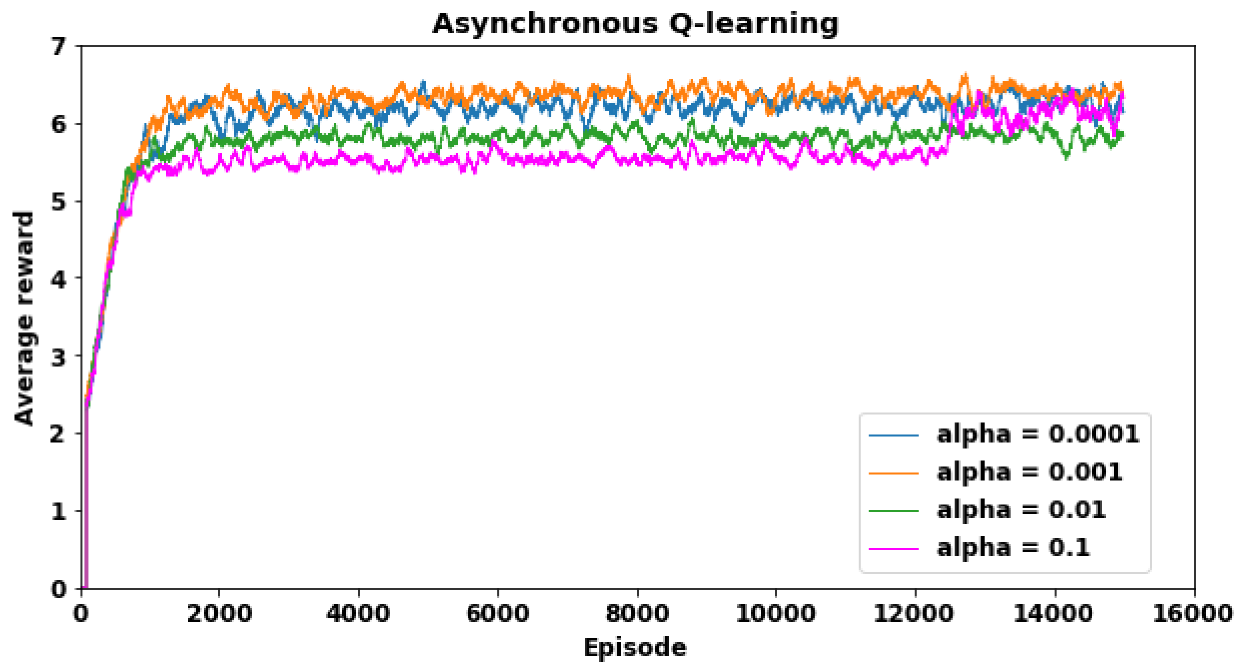

Fig. 6.3: Learning characteristics for the proposed asynchronous Q-learning

Fig. 6.4 shows a plot of the learning curves for both algorithms with the value of alpha set to 0.001. At this value of alpha, both algorithms converge to their highest average reward. It can be seen that the asynchronous Q-learning algorithms converges within fewer iterations and achieves higher average episodic total rewards than the conventional Q-learning method. The asynchronous Q-learning method converges at slightly above 6.0 while the conventional method converges at a value slightly above 5.0.

The asynchronous Q-learning algorithm displayed lower sensitivity to the learning rate and converged to higher total rewards compared to the conventional Q-learning method because it had its action space restricted. All the possible actions were limited to only a set of actions that meet the power balance constraint. The results of this test on the effect of learning rate imply that limiting the control actions of the agent to just those that are within the system constraints improves stability with respect to the sensitivity to the learning rate.

It can be deduced that the sensitivity of the Q-learning algorithms is indicative of how accurate each Q-value estimate is. If the action space is not properly constrained, then Q-value estimates become less accurate. Consequently, the algorithm becomes prone to oscillations even for very low values of learning rate as seen with the conventional Q-learning. In the asynchronous Q-

learning, the action space was properly constrained. Thus, the Q-value estimates were more accurate, and the learning trajectory was stable for a wide range of learning rates.

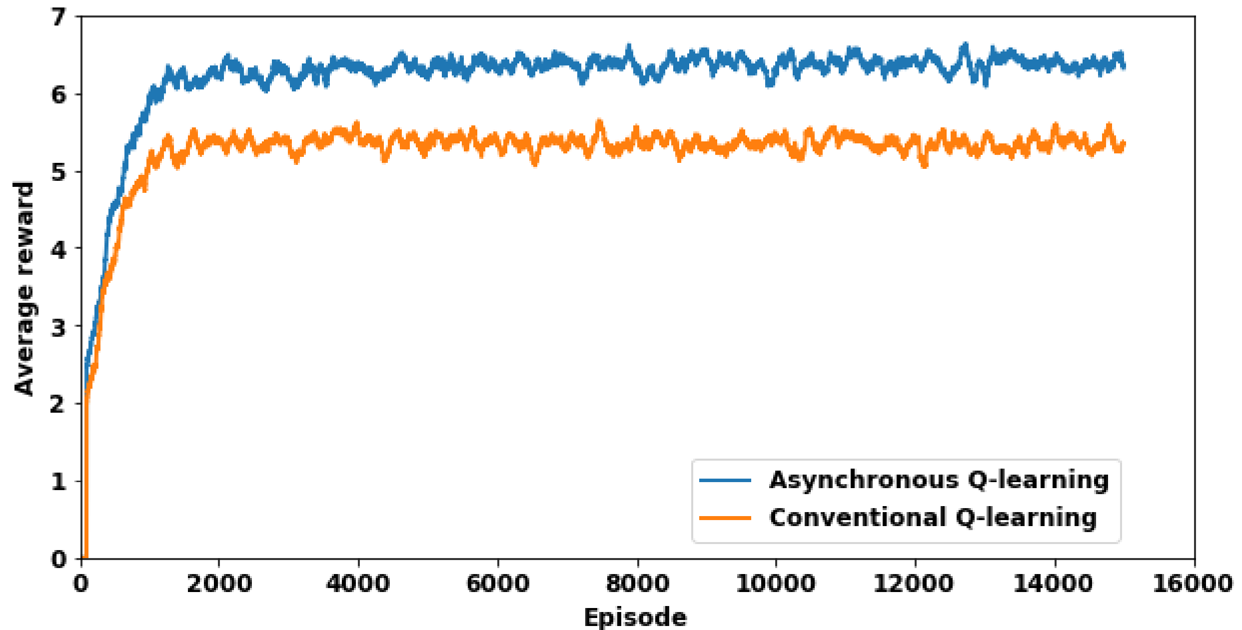

Fig. 6.4: Learning curves for the asynchronous Q-learning and the conventional Q-learning

6.3.3 **Effect of the Reward Signal on Q-learning**

Reinforcement learning does not optimize the cost directly. The cost optimization is implied by maximizing a reward signal. In this section, the effect of the reward function is explored. Particularly, the impact of the sign of the reward values produced by the reward function is investigated.

Intuitively, and as conventionally done, having the reward as negative of the cost is supposed to produce good learning outcomes because maximizing such a reward would naturally lead to a cost minimization. Such a reward function returns negative reward values; thus, it is hereinafter referred to as "negative reward". For the Q-learning algorithms investigated in this dissertation, the reward was defined as the inverse of the square of the cost plus a regularization constant of 1. The reward function returns positive reward values; thus, it is hereinafter referred to as "positive reward".

A test was done to explore the behaviour of the asynchronous Q-learning algorithm and the conventional Q-learning method under the two reward functions. A moving average episodic cost profile obtained using each of the reward functions was plotted on the same axis for both

algorithms. The moving average cost profile is a better way to compare the reward functions than the reward profile because the same objective function is used to calculate the cost in all the cases under consideration.

Fig. 6.5 and Fig. 6.6 show the episodic cost profiles for the conventional Q-learning and the asynchronous Q-learning methods correspondingly, for 15000 iterations under the various reward functions. It can be seen from the figures that the positive reward function produced a more stable learning curve than the negative reward function in both Q-learning algorithms. There was no marked difference in the learning behaviour of the two algorithms under the same reward function except for the fact that the asynchronous Q-learning returned a lower global cost (the difference in the global costs are explored in the next section).

The learning behaviour of both the Q-learning algorithms under the negative reward function was found to be unstable. This is because the Q-values were initialized by zeros while the new estimates are negative. Actions that have not been chosen always had higher Q-values than the ones that had been selected. Consequently, the agent is forced to explore the search space even when the action selection algorithm (the epsilon greedy strategy) recommended exploitation. This results in unstable learning and leads to poor convergence.

The learning curves of both Q-learning algorithms under the positive reward were found to be more stable and had better convergence than with other reward functions. This was because the reward function produced positive reward values. Actions that have been selected had higher Q-values than those that had not been chosen. As a result, the algorithm's learning is well guided by the exploration strategy employed.

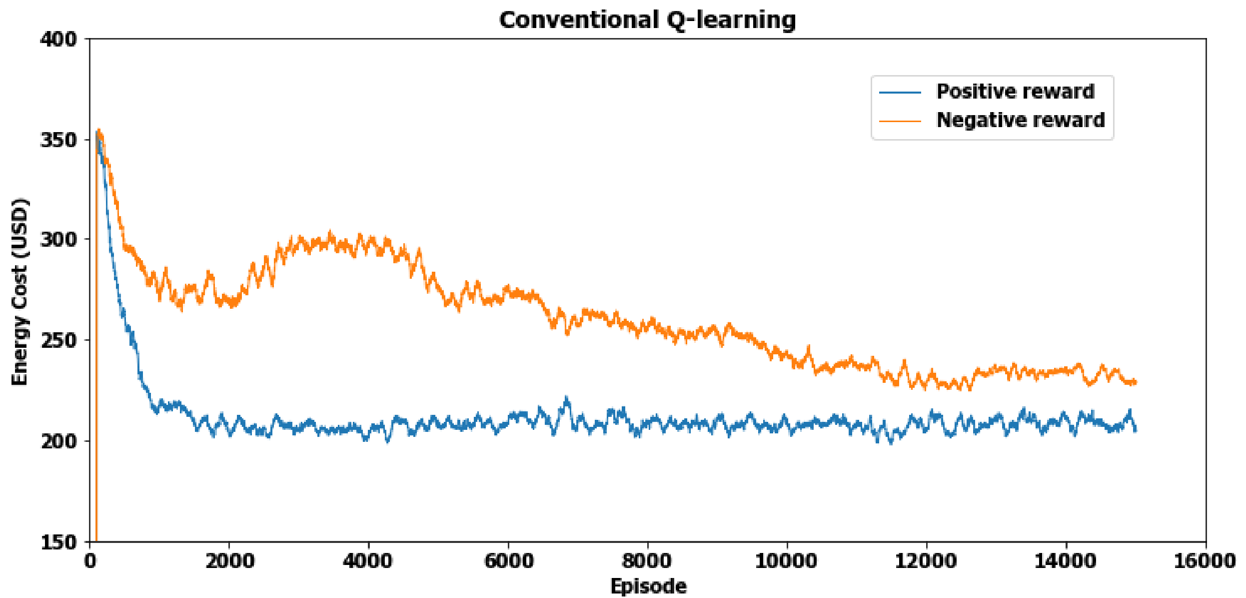

Fig. 6.5: Episodic moving average cost profiles for the conventional Q-learning under different reward functions

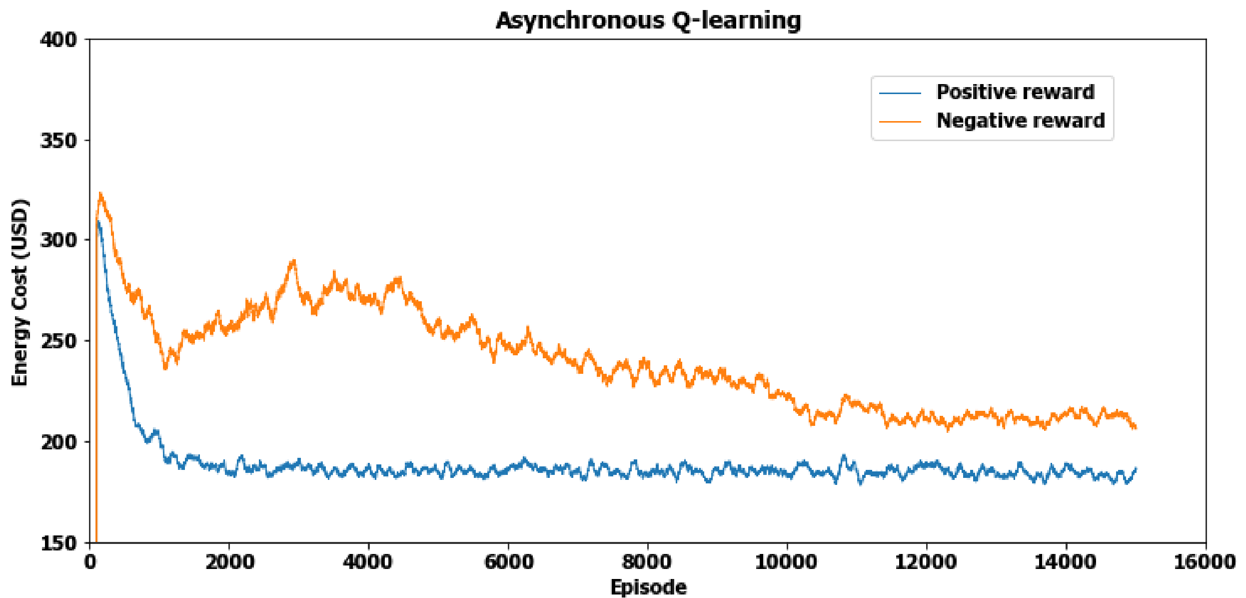

Fig. 6.6: Episodic moving average cost profiles for the asynchronous Q-learning under different reward functions

6.3.4 Global Cost Convergence of Q-learning

Fig. 6.7 and Fig. 6.8 show the cost profile for the conventional Q-learning and the asynchronous Q-learning algorithms correspondingly, for 15000 learning episodes. The simple rolling average

values are computed after every 100 iterations. It is evident in the two figures that the raw and average energy costs for the two methods start from high values in episodes from 0 to about 2000 and decline to lower values in the later episodes past 2000. This is because, in both algorithms, the original valued of epsilon was set at 1.0, i.e., the probability of selecting any of the actions was 1.0 at the beginning.

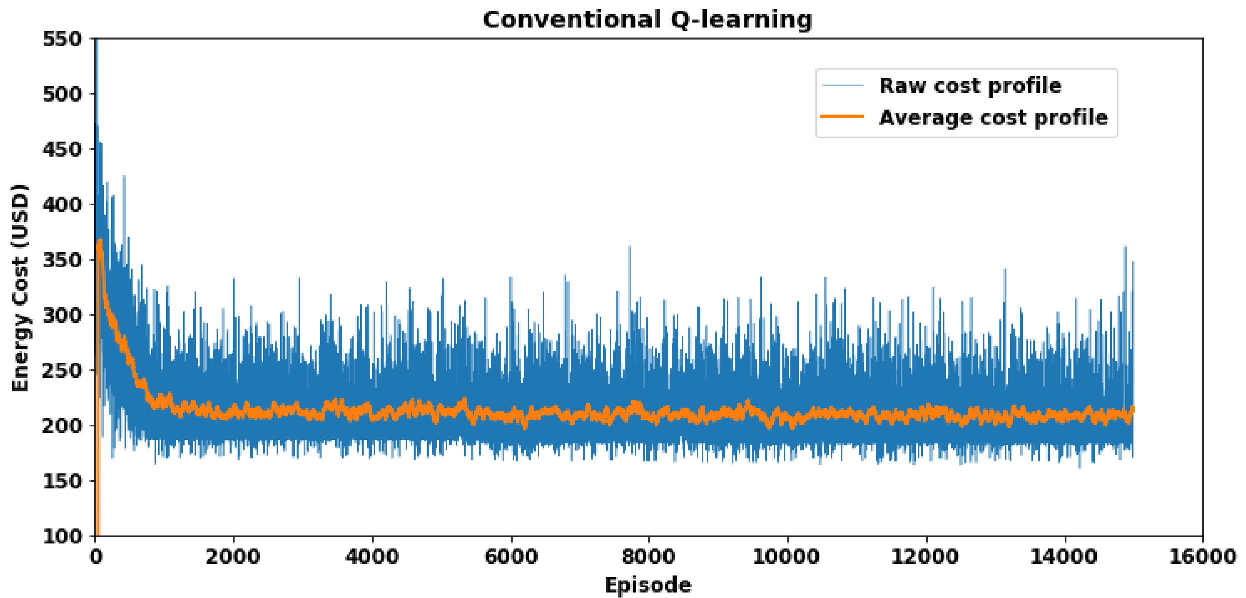

Fig. 6.7: Episodic cost profile for the conventional Q-learning method

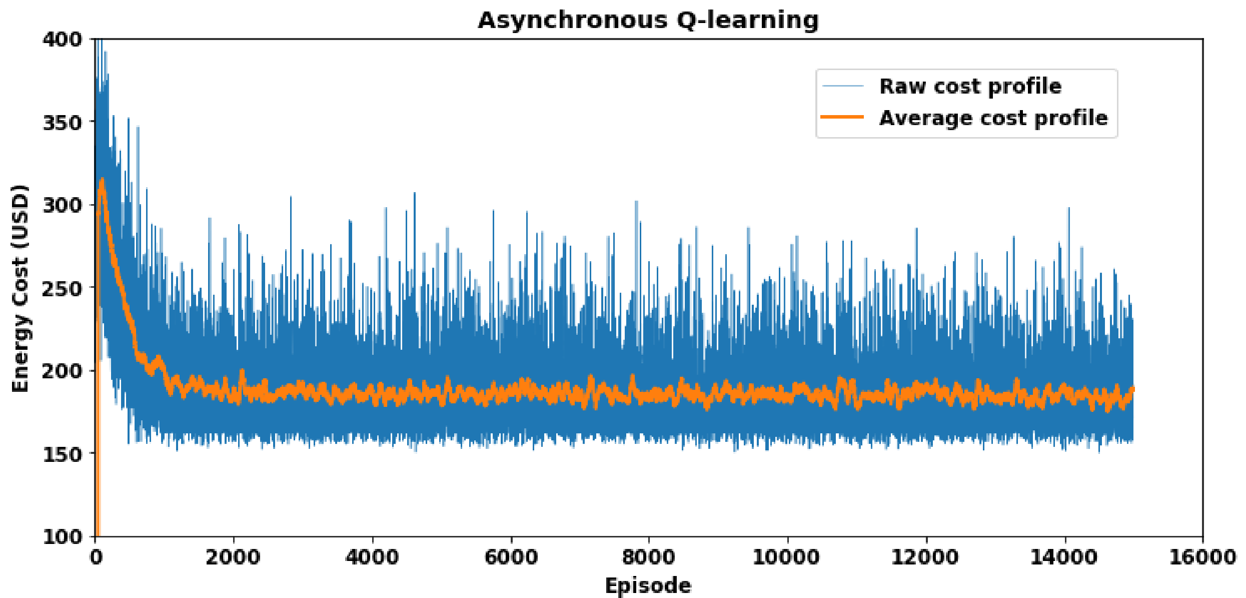

Fig. 6.8: Episodic cost profile for the proposed asynchronous Q-learning method

As the learning proceeds and the value of epsilon is gradually reduced, the actions that give higher rewards, and consequently, return lower costs in each time step are selected with higher probability. With the time-step action selection getting more inclined to the optimal actions, episodic global cost converges to the optimal value according to the temporal difference control theory. Both algorithms are shown to converge at their optimal values of the global cost.

Fig. 6.9 shows a plot of the moving average episodic cost profiles for both methods in the same graph. As can be seen in the figure, the asynchronous Q-learning algorithm begins the learning with lower initial moving average cost of about \$320 and completes the learning phase at a value of between \$200 and \$150 compared to the conventional Q-learning that starts from a higher value of just above \$350 and finishes at a value between \$250 and \$200. This is due to the fact that the proposed method has its action space properly constrained. Thus, the agent could only select actions that occur within the feasible set of actions determined by the power balance equation. Therefore, the power balance constraint is enforced before an action is selected. This effectively guides the learning process, thus, preventing the agent from losing track. That is unlike the conventional Q-learning approach that uses the battery scheduling formulation in which the power balance is imposed only after the agent takes the action. Therefore, the agent gathers bad transitions that cause it to lose track.

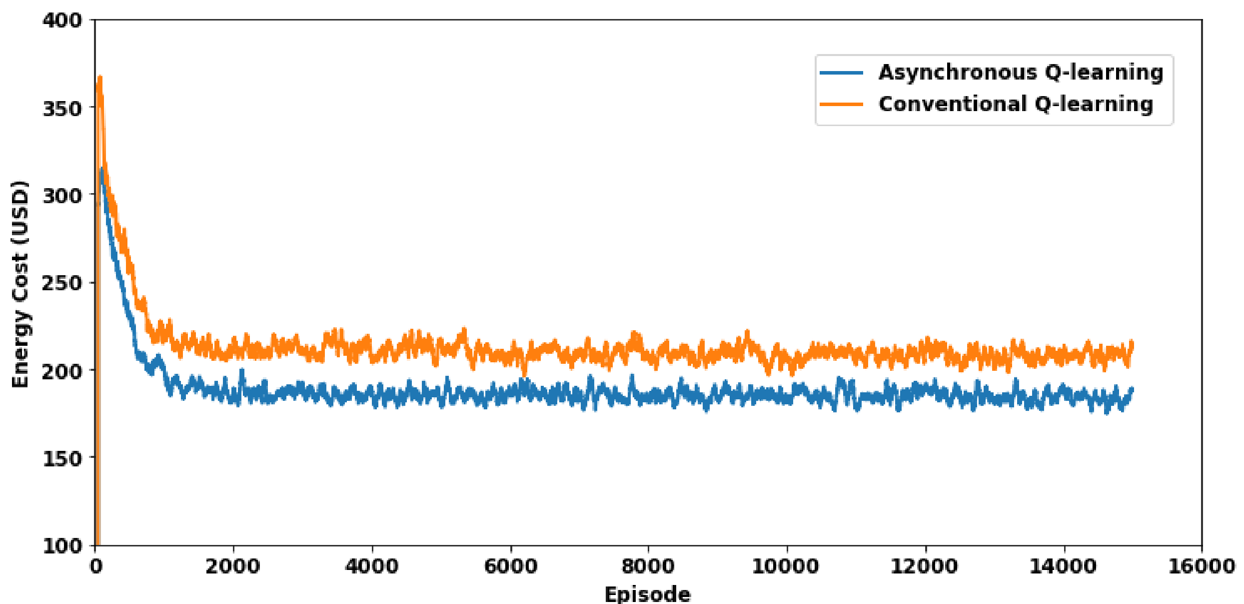

Fig. 6.9: Comparative cost profile for the asynchronous and the conventional Q-learning

Table 6. 3 shows the final global cost returned by the two algorithms when their respective policies were retrieved. The algorithms were trained to learn a policy that minimizes the global cost of charging EV throughout a 24-hour horizon. To retrieve the policy, the Q-table returned by the Q-learning algorithm is obtained. For each state, the action that has the highest Q-value is selected. The cost of the action is calculated. This is done for states occurring from time, t=0 to t=23. The optimal episode represents the power schedule returned by the algorithm. The optimal episode for the asynchronous Q-learning algorithm returned about 14% lower global cost than the conventional Q-learning.

It can be deduced that the difference in the final global costs is mainly caused by the difference in the battery usage under the two algorithms. The total battery degradation cost depends on the amount of stress the power schedule produced by the algorithm exerted on the battery. Battery degradation cost increases exponentially with the fractional change in the depth of discharge [21], [109]. It can be seen in Table 6.3 that the power schedule produced by the conventional Q-learning algorithm had a battery degradation cost of at least 4 times that of the asynchronous Q-learning.

Fig. 6.10 shows the optimized battery energy profiles for both the conventional and asynchronous Q-learning algorithms. It can be seen that the asynchronous Q-learning algorithm recommended shorter charge/discharge cycles than the conventional Q-learning method. The standard deviation (STD) of the battery energy profile returned by the asynchronous Q-learning over the 24 hours was found to be about 26% lower than that of the conventional Q-learning algorithm. Since the degradation cost increases exponentially with the size of the charge/discharge rates, a small difference in the rates causes a big difference in the degradation cost.

Table 6.3: Returned costs for both algorithms

Learning Algorithm	**Returned Global Cost (USD)**	**Cost of Grid Power Purchase (USD)**	**Battery Degradation Cost (USD)**	**STD of battery energy profile (kWh)**
Conventional Q-learning	188	150	38	22.6
Asynchronous Q-learning	162	153	9	16.9

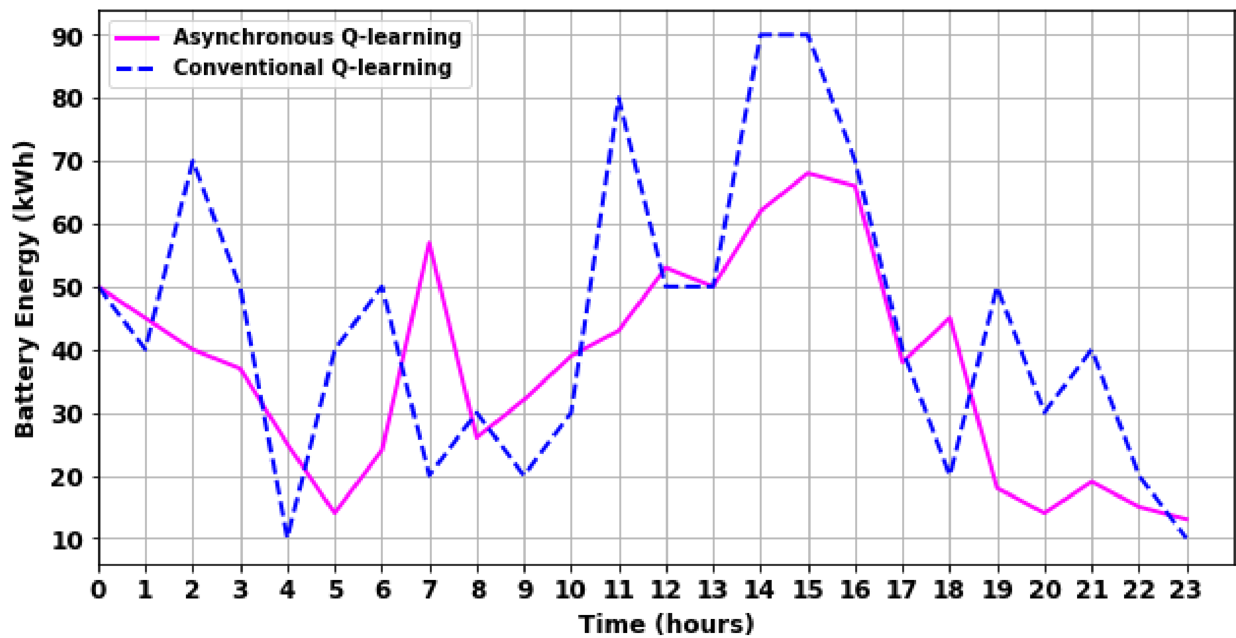

Fig. 6.10: Optimized battery energy profiles for both Q-learning algorithms

6.3.5 **Optimized Power Schedule: Asynchronous Q-learning**

When the learning phase is completed, the optimal power schedule is retrieved from the Q-table using a greedy policy. The greedy policy selects the action with the greatest Q-value in every state considering all the episodes of learning. The actions are then used to produce the power schedule for the 24 hours. In this section, the power schedule obtained using the proposed asynchronous Q-learning is discussed. We will identify and explain the actions recommended by the algorithm and explain how they led to minimization of the energy costs.

In understanding the schedule, the signs of the scheduled power values of the battery and the grid should be taken into consideration. Any scheduled power is always positive when supplying the load through the common bus and negative when drawn from the bus. A negative battery power, therefore, represents a charge command while a positive battery power means a discharge command. Therefore, the algorithm ensures that charge and discharge operations cannot occur at the same time without the need for a secondary controller to enforce that constraint. Similarly, a negative grid power denotes a power supply to the grid whereas a positive grid power means a power purchase from the grid. Also, the battery energy transitions are occasioned by power schedule actions, thus the effect of a battery charge or discharge command in a time step is observed in the next time step.

The algorithm was design to learn a policy (strategy) that when used to select a power schedule leads to a minimization of the cost of meeting the load at the charging station. Intuitively, the following strategies could lead to cost minimization:

i) Maximizing the self-consumption of the PV generated power.
ii) Using less grid power when the tariff is high and more of it when the tariff is low.
iii) Performing short charge/discharge cycles to lower the battery degradation cost.

Fig. 6.11 shows the optimized power schedule produced by the proposed asynchronous Q-learning algorithm. For ease of analysis, the battery energy values in kWh were divided by the timestep (1 hour) to get kW values and plotted on the same axis as the load, PV and grid power.

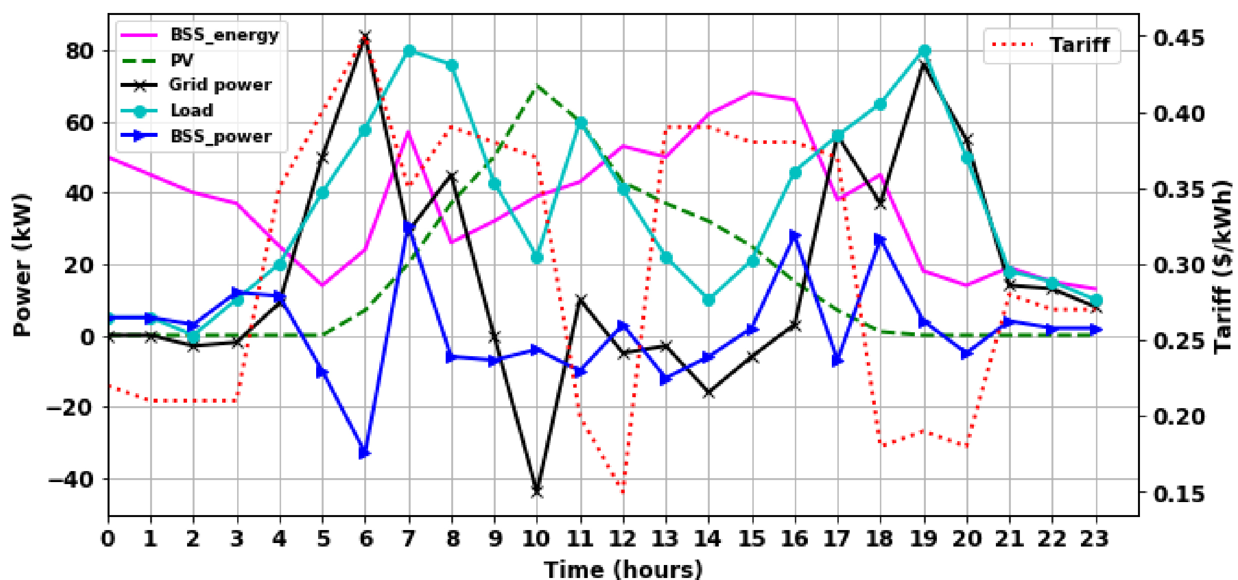

Fig. 6.11: Optimized power schedule obtained using the asynchronous Q-learning algorithm (battery energy values in kWh divided by time to get kW values)

The first policy is executed by charging the BSS when the solar PV generation is high and using the stored energy when the PV is insufficient, and the grid power is expensive. The battery power and grid power are positive when feeding power into the DC bus and negative when absorbing power from the DC bus. It can be seen in Fig. 6.11 that the general battery energy trend shows a discharge from the 0 to 5th hour (see the magenta curve labeled "BSS_energy"). From the 5th to the 7th hour, the battery charges as PV generation rises. However, a discharge occurs from the 7th to the 8th hour to supply the load that peaks at around this time. From the 8th to the 15th hour, the battery continues to charge due to high PV generation, except for a small instability at the 12th

hour, which is expected due to the stochasticity in the environment causing some suboptimal decisions to be made by the agent. From the 15th to the 20th hour, the battery shows a discharge trend due to the drop in PV power and an increasing load to be supplied. Again, there is a suboptimal decision to charge at the 17th hour.

The second strategy is implemented by occasioning high grid power intake when it is cheap and reducing the purchase of grid power when it is expensive. Also, cost savings are achieved by generally minimizing grid power intake since the grid power cost is mostly higher than the cost of battery degradation for small power values. This, however, depends on the existing load to be supplied and the availability of PV power and the energy level in the battery. In Fig. 6.11, it can be seen that there is a high grid power intake between the 5th and the 7th hour despite the high grid tariff because, at this time, the battery energy is almost completely drained, there is low PV generation, thus the need to use the grid to supply the rising load (i.e., the black curve labeled "grid"). The grid power intake is generally low between the 9th and the 16th hour, even going to negative at the 10th hour when the grid absorbs power, due to high PV generation. At the 10th and 12th to 15th hour, the station supplies power back to the grid due to excess PV generation. However, power is not supplied to the grid in the 11th but is drawn from it due to low tariff, which implies that the utility grid is producing more power than the load demand at that time.

The third strategy is implemented by limiting the size of the charge/discharge steps. Although it is expected to be more cost effecting to use all the excess PV to charge the battery, the inclusion of battery degradation cost limits the charging rate. It is therefore costly to recommend high charge or discharge schedules since the cost of battery degradation due to depth of discharge grows exponentially with an increase in the magnitude of charge/discharge steps [105]. Therefore, the algorithm generally recommends shorter charge and discharge cycles to protect the battery. It is in this regard that the two algorithms differ in their global costs.

6.3.6 Scalability of the Q-learning Algorithms

Scalability of an algorithm may be viewed as its capacity to retain the computational speed or efficiency when the dimensions of the problem is increased. Q-learning algorithms use a look-up table (the Q-table) to store Q-values for state-action pairs. The size of the Q-table depends on the state-action space. An increase in the state-action space increases the size of the Q-table. Consequently, the computational time increases with the increase in the size of the Q-table. This

is famously referred to as the curse of dimensionality. It limits the scalability of Q-learning algorithms. This section investigates the scalability of the Q-learning algorithms.

To visualize the scalability of the two Q-learning methods, the computational time of the algorithms was recorded for different sizes state-action spaces. The state-action space may be changed by changing the discretization steps of any of the state variables or the timestep sizes. The state variables included the load, the PV output, the grid tariff and the battery energy level. There was no data on the first three variables for smaller steps. Therefore, the battery energy was chosen as the basis of changing the state-action space. The battery size was increased from a value of 100 kWh to 300 kWh in steps of 50 kWh. The changes in the battery capacity was taken to be representative of the changes in the state-action space. Both algorithms were run under the various values of the battery capacity and the training time was recorded in each run. For each value of battery capacity, the initial battery energy was set at half the battery capacity for consistency.

Fig. 6.12 and Fig. 6.13 show the plots of computational time against battery size for the conventional and the asynchronous Q-learning respectively. The trends of the scalability curves show that the computational time is proportional to the size of the state-action space for both algorithms. If the system size is big, i.e., if a larger battery is used, the algorithms would take longer to train.

In comparison, the conventional Q-learning is more scalable than the asynchronous Q-learning. On average (when the average slopes of the curves were computed), increasing the battery size by 50 kWh led to an increase in computational time of 0.25 seconds in the conventional Q-learning and 15.15 seconds in the asynchronous Q-learning. Since scalability is the capacity of an algorithm to retain the computational speed when the dimensions of the problem are increased, we can take the inverse of the changes in computational time per 50 kWh increase in battery size as a measure of scalability. Thus, the scalability of the conventional Q-learning was found to be 3.96 while that of the asynchronous Q-learning was found to be 0.07. That means that the computational speed of the conventional Q-learning is about 56 times more resilient to increase in the dimensions of the problem than the asynchronous Q-learning. This is because for the asynchronous Q-learning method, each state and its list of possible actions were added to the Q-table. This was done even though some of the states had some of their possible actions being the same. Consequently, there were redundancies in the Q-table. The redundancies caused a significant increase in the

computational time with increase in system size. The conventional Q-learning on the other hand has a common action space for all states.

It can also be observed that the vertical axis values for the asynchronous Q-learning algorithm are bigger than those for the conventional Q-learning algorithm. Considering the battery size of 100 kWh, the asynchronous Q-learning algorithm trained for 130.1 seconds (first plotted point in Fig. 6.13) compared to 4.8 seconds for the conventional Q-learning algorithm (first plotted point in Fig. 6.12). The conventional Q-learning was about 27 times faster than the asynchronous Q-learning method. Therefore, it can be said that although the restriction of the action space in every state in Asynchronous Q-learning led to a convergence to a lower global cost, the benefit came at the expense of scalability and computational speed.

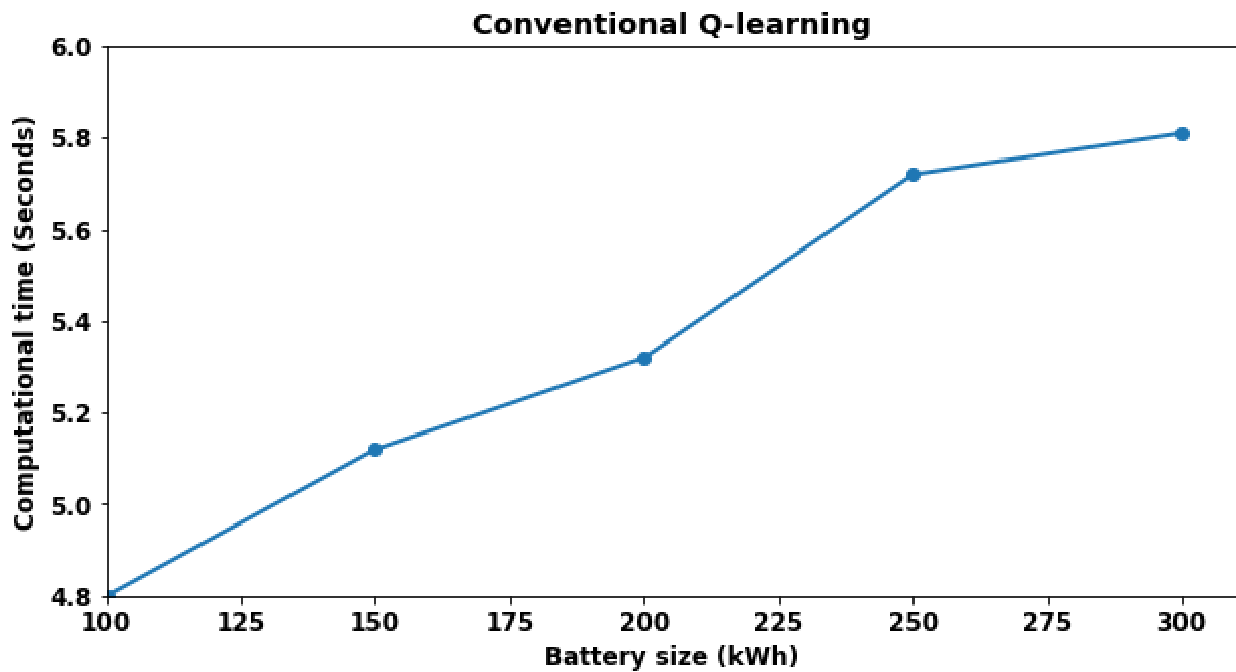

Fig. 6.12: A plot of computational time against battery size for the conventional Q-learning

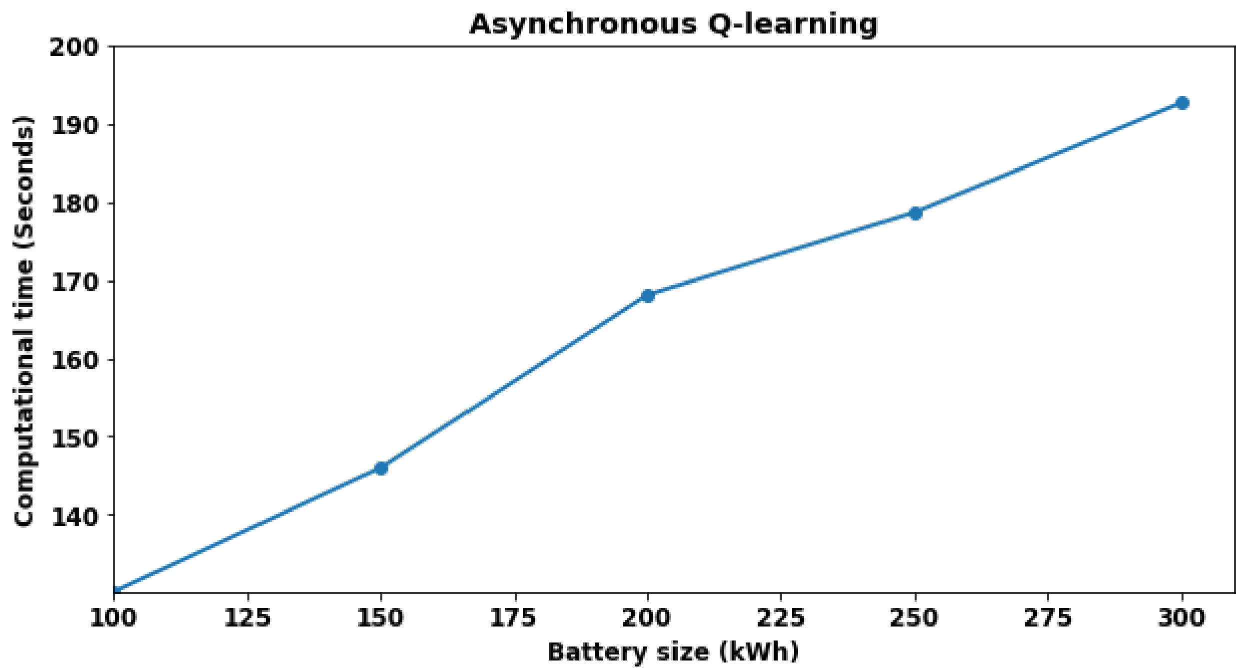

Fig. 6.13: A plot of computational time against battery size for the asynchronous Q-learning

6.4 Simulation Results of A2C Algorithm

6.4.1 Learning Parameters

Table 6.4 shows the learning parameters chosen for the A2C algorithm. For better performance, the critic learning rate (α) is normally set at a slightly higher value relative to the actor network learning rate (β) [93]. Different learning rates for this architecture have been tested for values in the order of between 10^{-2} and 10^{-4}, and have been found to produce fairly good results [110]. Further tuning was done through trial and error to arrive at a β of 0.0001 and an α of 0.0003. The discount factor was chosen to be 1.0. This value ensures that current and future rewards have equal value in the present.

The step size (D) determines the number of transitions to be gathered before an update is done. In the A2C framework, the higher the value of D, the higher the variance in the episodic reward profile and the poorer the convergence in the learning curve [111], [112]. Conventional actor-critic does not gather any experiences before performing an update. The updates are done after every transition; thus, the value of D is 1 by design of the algorithm. To select the value of D for A2C, the number of transitions in an episode in this problem was considered. An episode in this problem has 24 transitions. Values of D closer to 24 (full episode) produce very high variance with slightly

lower global cost than values below half of 24. To reduce the high variance, a value of 10 was chosen.

Table 6.4: A2C learning parameters

Learning Parameter	Symbol	Chosen value
Actor learning rate	β	0.0001
Critic learning rate	α	0.0003
Step size	D	10
Discount factor	γ	1.0
Number of episodes	n	5000

6.4.2 Learning Characteristics and Cost Convergence

The A2C algorithm was run for 5000 episodes. Also, the conventional actor-critic (AC) was also run with same learning parameters and actor and critic network models. The only difference was that the A2C employed experience replay and learning parallelization which are not found in the design of the AC algorithm.

Fig. 6.14 and Fig. 6.15 show the learning curves for the AC algorithm and the A2C algorithms respectively. Comparatively, the conventional AC algorithm converge within fewer iterations and has lower variance in the learning curve. The AC converged after less than 500 iterations while the A2C converged after slightly more than 500 iterations. The AC algorithm converged earlier than the A2C because the updates in AC were done after every transition. In an episode of 24 transitions, the AC was updated 24 times while the A2C gathered 10 transitions before performing an update. Also, the variance in the raw reward profile for the AC was lower than that in the A2C raw reward profile. The standard deviation (STD) of the raw rewards in the AC was found to be 17. 3 while that for A2C was 50.9. That means that the A2C had about 3 times higher variance in the learning trajectory than the AC algorithm. The variance is higher in A2C because more steps are considered before performing an update. On the other hand, low variance is observed in AC because the updates are done at the end of each transition. However, the low variance in AC

algorithm comes with a high bias. The AC has a high bias because its networks are updated after every transition. Thus, the estimates by the critic and the actor networks drive the agent to an early convergence before enough experiences are collected to make the decision as to whether the point of convergence is a global optimum. It can be seen in Fig. 6.14 that the convergence of the AC as seen from the average reward profile after 1000 episodes is almost a straight line with much fewer oscillations in the raw reward profile than the A2C profile shown in Fig. 6.15. That is due to the high bias in the algorithm. High bias prevents the algorithm from exploring the search space, thus, it increases the risk of the algorithm being trapped in a local optimum. Although the variance is higher in A2C, the algorithm explored the search space better and converged to a lower global cost than the AC algorithm. That exploration may be attributed to the use of experience replay and parallelization of learning in A2C. These techniques introduce some noise in the policy that prevents the algorithm from being trapped in a local optimum. To visualize the difference in the global cost, the moving average reward profiles for the two algorithms were plotted on the same axis.

Fig. 6.16 shows the average reward profiles of the two algorithms plotted on the same axis. Episodic cost values are derived from the negative reward values by multiplying the reward values by -1. It is evident in Fig. 6.16 that the A2C algorithm converged to a lower global cost than the AC algorithm. This is despite the high variance in the learning curve. The final global cost for A2C was found to be $155 while that of AC which was found to be $179. It can be deduced that parallelization of learning and experience replay in A2C improves convergence.

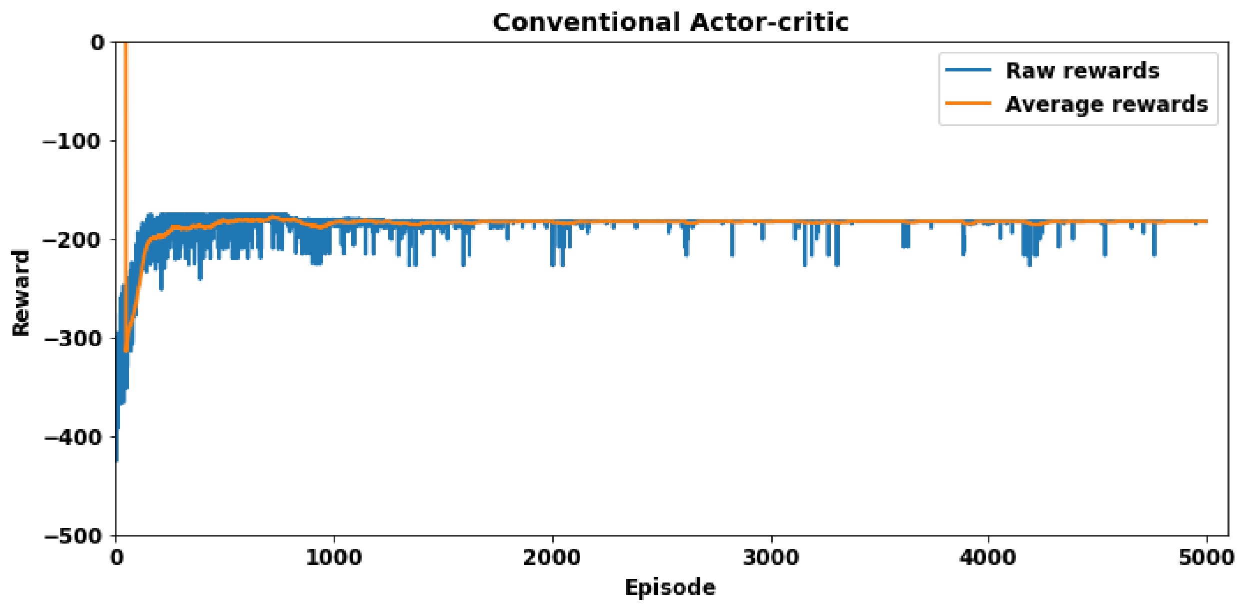

Fig. 6.14: Learning curve for the conventional AC algorithm

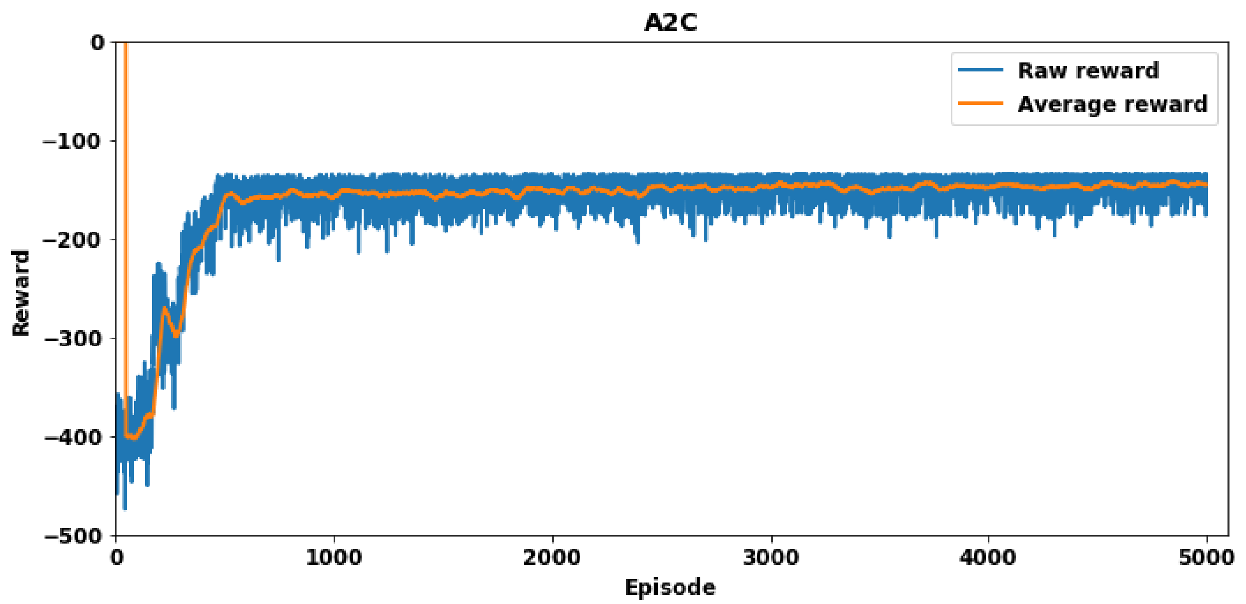

Fig. 6.15: Learning curve for the A2C algorithm

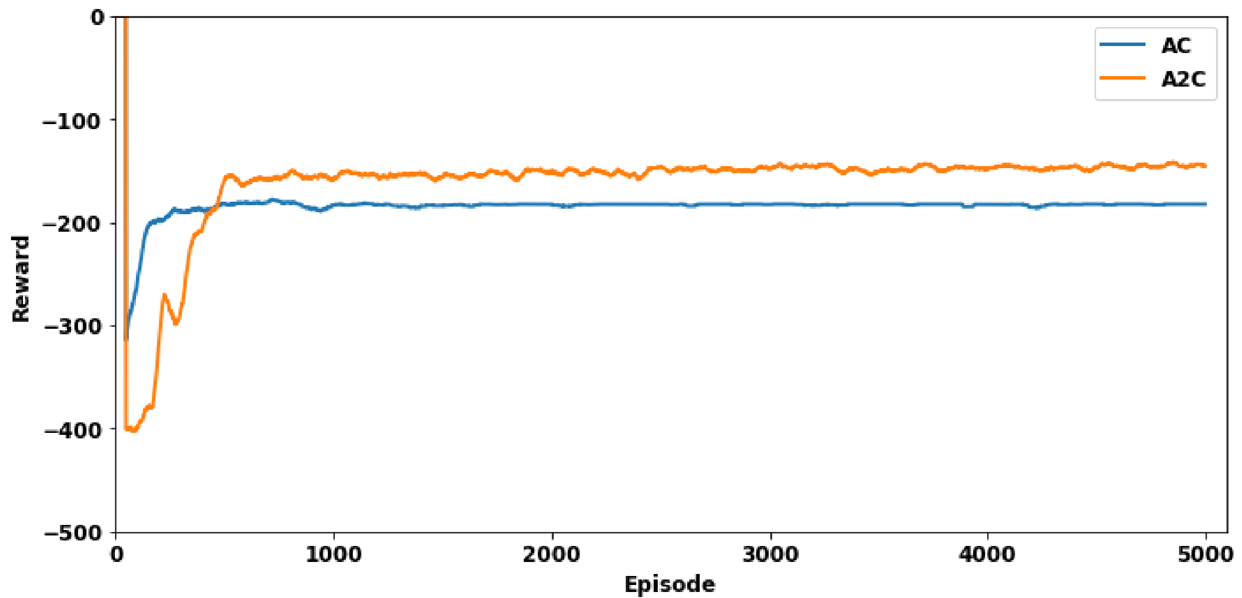

Figure 6.16: Episodic reward profiles for the conventional AC and the A2C algorithms

6.4.3 Impact of reward function on A2C and AC

To visualize the performance of the reward functions on the A2C, a plot of the average cost profile for the algorithm under the two reward functions, namely, negative reward and positive reward as described Section 6.3.3 was included.

Fig. 6.17 and Fig. 6.18 show the episodic cost profiles returned by the AC and A2C algorithms respectively under the two reward functions. It can be observed that both algorithms returned a more stable learning curve with the negative reward function than with the positive reward function. Also, the figures show that the negative reward function produced a convergence to a lower global cost than the positive reward function in both algorithms. Unlike Q-learning that uses Q-values to learn the cost minimization policy, the AC and the A2C algorithms parameterize the policy directly using the policy gradient theorem. It was established that a negative reward function counteracts the influence of the exploration strategy (epsilon greedy strategy) employed for the purpose of action selection in the Q-learning algorithms. However, in the actor-critic architecture, there is no separate exploration strategy. The action selection is done using the policy returned by the actor network. The policy in this case is a probability distribution by which the actions are to be selected. The A2C algorithm learnt more naturally and stably under negative reward function. Thus, the negative reward function was chosen for this algorithm.

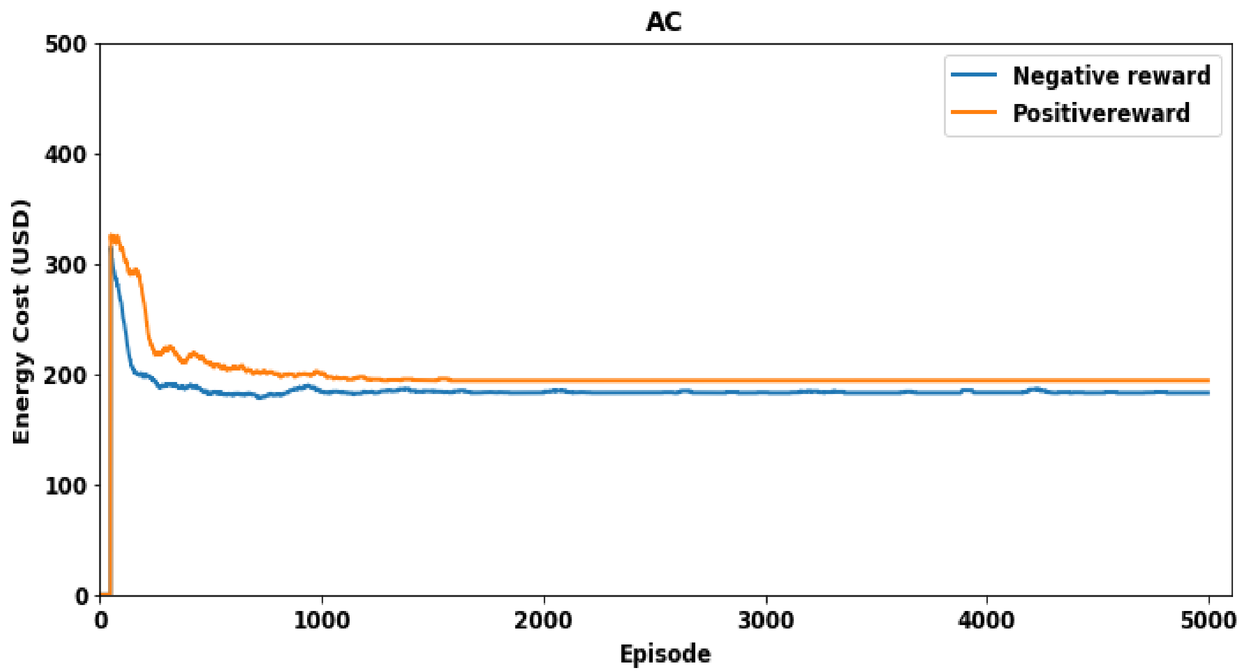

Figure 6.17: Episodic cost profile for conventional AC algorithm under different reward functions

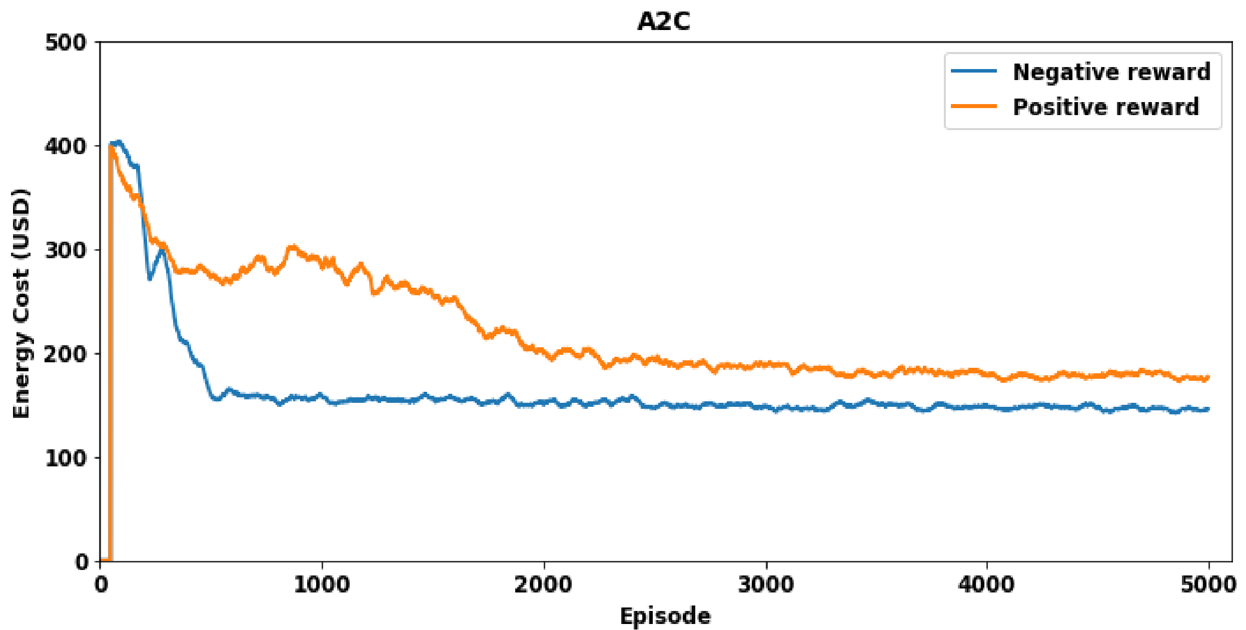

Fig. 6.18: Episodic cost profile for A2C algorithm under different reward functions

6.4.4 **Optimized Power Schedule: A2C**

In this section, the cost minimization policies learnt by the A2C algorithm are identified and explained. The cost of energy needed to charge the electric vehicles included the cost of buying

power from the utility and the cost of degradation of the BSS. Fig. 6.19 shows the optimized power schedule obtained from the A2C algorithm under static optimization conditions.

To minimize the cost of purchasing power from the grid, the algorithm learnt to increase the intake of power from the grid when the grid tariff is low. In Fig. 6.19 it can be seen that the grid power intake (see the black curve) was high between the 10th and the 12th hour and between the 17th and the 21st hour when the grid tariff was low. However, when the load at the charging station is high and the PV generation is inadequate to meet the demand, then the algorithm schedules the grid power purchase even though the grid tariff is high. From the 3rd hour the grid power is seen to increase with the load until the load peaks at the 7th hour and decreased with the load upto the 8th hour. As the PV generation increased with a reducing load, the grid power decreased to negative at the 9th and the 10th hour. A negative grid power value means that the grid was taking in power from the station. Also, the algorithm learnt to decrease the grid power intake when the tariff is high. In Fig. 6.19, low grid power intake was observed between the 13th and the 16th hour. Furthermore, the algorithm learnt to maximize the self-consumption of the PV generated power. The battery profile (see the magenta curve labelled "BSS_Energy") showed a general discharge when the PV was low and a charge when the PV was high. Discharge can be seen from the beginning to the 6th hour with some instances when the battery goes to idle mode. When PV generation starts to increase from the 6th hour, a discharge trend is observed from this time. However, the algorithm recommended a discharge from the 8th to the 10th hours, which is a suboptimal action. Such suboptimal actions are expected in a stochastic environment where the policy of cost minimization being learnt is also stochastic. It was noted that the battery did not charge to its maximum value of 100 kWh. This is because the PV was insufficient. Also, the algorithm recommended small charge rates that could not drive the battery to full charge within the time when the charging occurred. When PV generation went low in relation to the load, the battery discharged to supply the load as seen from the 16th to the 18th hours. From the 18th hour, limited battery chargeing occurred from grid power due to low grid tariff between 18th and 20th hours. When the grid tariff rose again, the battery discharged to supply the load as the PV generation had gone to zero.

To minimize the cost of battery degradation, the algorithm learnt to recommend short charge and discharge operations. A comparative plot of the battery energy profiles for the A2C and the AC algorithms was included as shown in Fig. 6.20. It can be seen that the A2C algorithm recommended

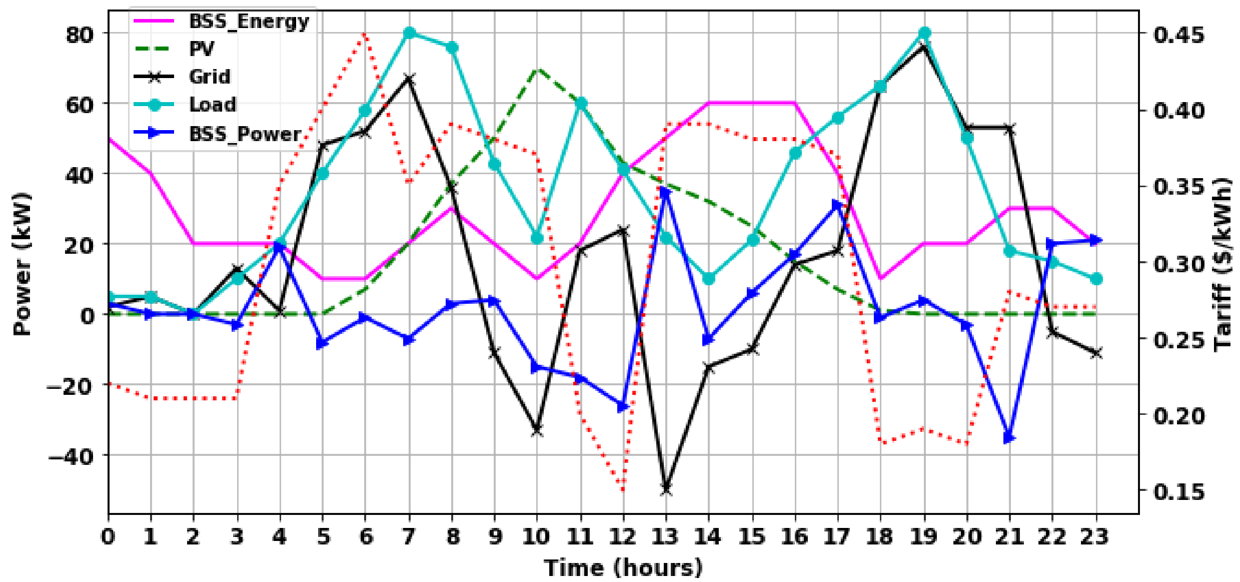

Fig. 6.19: Optimized power schedule obtained from the A2C algorithm ((battery energy values in kWh divided by time to get kW values)

shorter charge and discharge cycles compared to the AC algorithm, thus, it returned a 13.4% lower global cost than the AC method. The length of the charge/discharge cycles can be traced by the variations in the battery energy levels from the mean value. Table 6.5 gives a summary of the final global costs, battery degradation costs and the STD of battery energy profiles returned by the two algorithms. The STD of the battery energy profile for the A2C was 16.0 kWh while that for the AC was 20.9 kWh. Consquently, the battery degradation cost returned by the schedule obtained using A2C was more than 3 times lower than that returned by the AC power schedule. Although the difference in the STDs for the battery energy profiles was small, the degradation cost difference between the two algorithms was quite significant. This is because battery degradation increases exponentially with the size of the charge/discharge cycles.

Table 6.5: Global costs, battery degradation cost and the STD of battery energy profile for A2C and AC

Algorithm	**Global Cost (USD)**	**Grid power purchase cost (USD)**	**Battery degradation cost (USD)**	**STD of Battery Energy Profile (kWh)**
AC	179	151	28	20.9
A2C	155	147	8	16.0

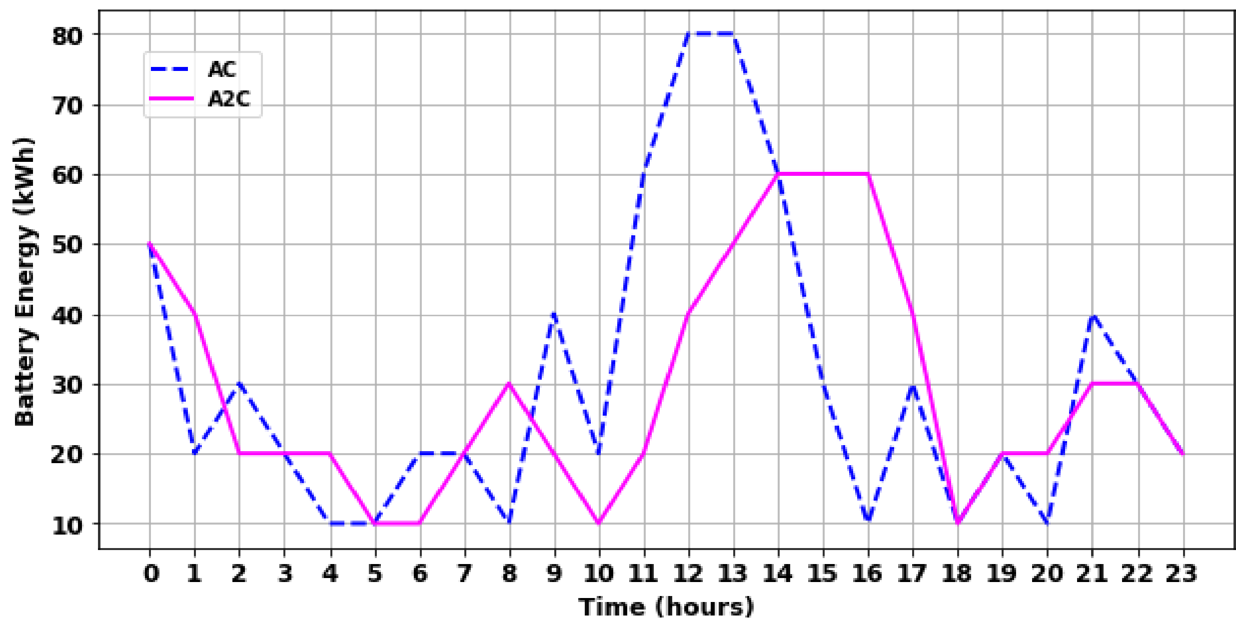

Fig. 6.20: Optimized battery energy profiles for both the A2C and the AC algorithms

6.4.5 Scalability Analysis for A2C

Unlike Q-learning, actor-critic algorithms do not use a lookup table to store the states and possible actions in order to track the learning. Instead, an ANN-based actor-critic algorithm uses its weight matrix to an ANN to predict both the policy (actor network) and the value function (critic network). That is why actor-critic algorithms can be used for problems with continuous state spaces. This section investigates the scalability of the A2C algorithm in comparison to the conventional AC method.

To investigate the scalability of the actor-critic algorithms, the state-action space is increased and the time it takes to return the final solution is recorded. Due to lack of data on smaller discretization of the state variables (load, PV generation and grid tariff), this study will consider increasing the action space of the problem by increasing the battery size. The battery size was increased from 100 kWh to 300 kWh in steps of 50 kWh and the algorithmic computational time was recorded. The battery discretization step was kept constant.

Fig. 6.21 and Fig. 6.22 show the plots of computational time against battery size for the conventional AC algorithm and the A2C algorithm respectively. It can be seen from the figures that the computational time increased with an increase in the battery size. This is because the battery size determined the number of nodes in the output layer of the actor network provided the

battery discretization was kept constant. An increase in the number of nodes increases the computational time of the actor-critic algorithm.

Comparing Fig. 6.21 and Fig. 6.22, it can be observed that the A2C algorithm is more scalable than the AC algorithm. On average, for every 50 kWh increase in the battery size, the A2C computational time increased by 0.1175 seconds while that of AC increased by 6.99 seconds. Consequently, the scalability (taken to be the inverse of the change in computational time per 50 kWh increase in battery size) of A2C was 8.51 while that of the AC was found to be 0.14. That is, with respect to the measure of scalability used in this study, the A2C algorithm was found to be about 60 times more scalable than the AC algorithm. This difference in scalability may be attributed to the number of updates done by the algorithms per episode. In each episode of learning, the A2C performed only about two updates. This translates to slightly over 10000 updates in 5000 episodes. On the other hand, the AC algorithm performed up to 24 updates. This translates to about 120 000 updates in the 5000 episodes. The AC algorithm performed at least 12 times more updates than the A2C algorithm. An update is done through the backpropagation algorithm. The algorithm calculates the gradient of the loss function with respect to each element in the weight vector of each neural network starting from the output layer backwards. The objective of each update is to minimize the loss. This update process consumes time. Thus, if more of the updates are done, the computational time is higher. Performing fewer updates per episode makes the A2C algorithm faster and more scalable than the conventional AC algorithm. For instance, for a battery size of 100 kWh, A2C computed the optimal results in about 53 seconds while the AC algorithm took about 282 seconds. That means the A2C was 5 times faster than the AC algorithm.

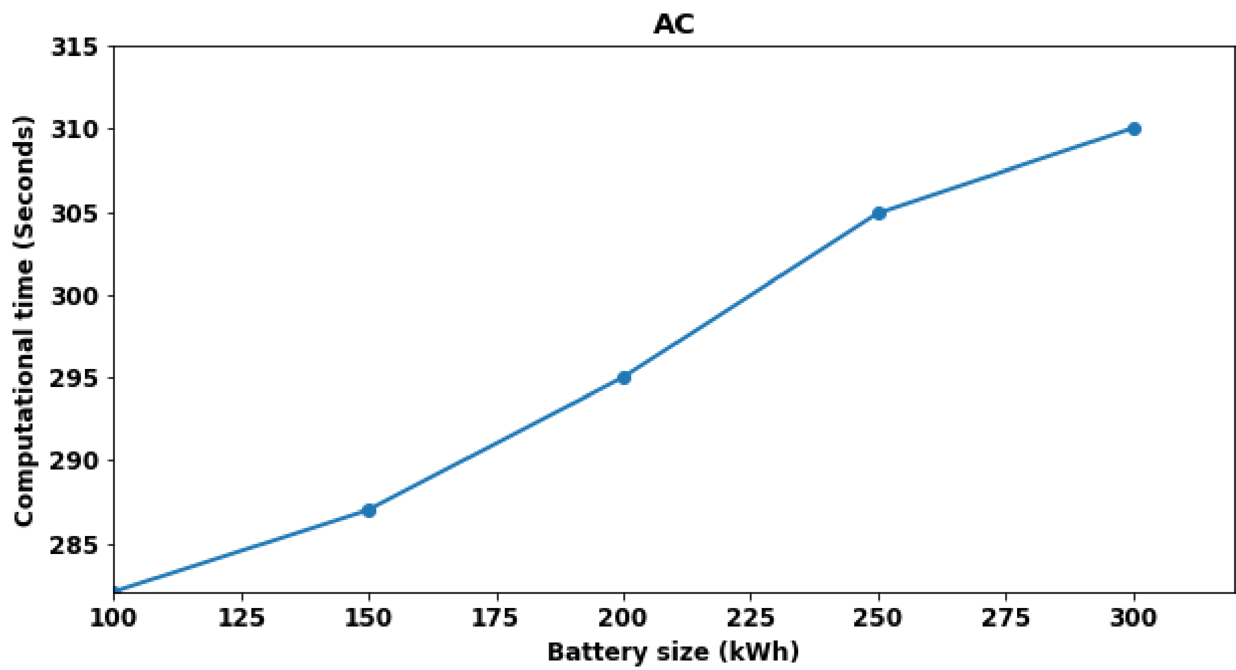

Fig. 6.21: A plot of computational time against battery size for the conventional actor-critic algorithm

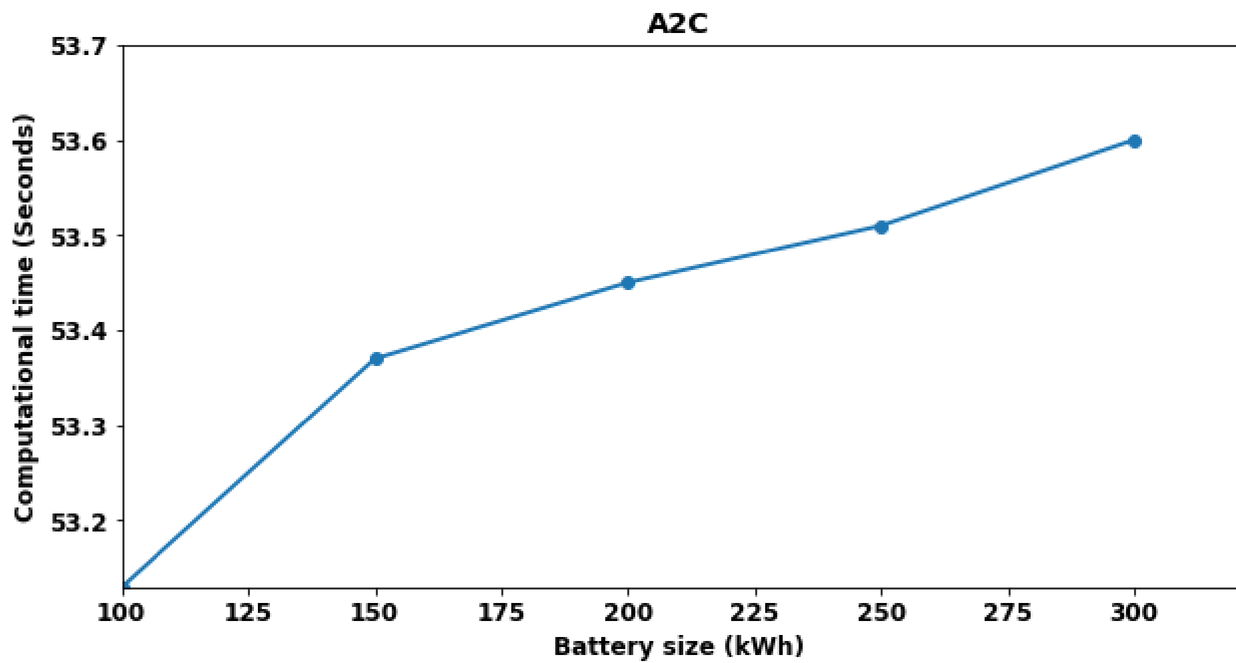

Fig. 6.22: A plot of computational time against battery size for the A2C algorithm

6.5 Discussions

6.5.1 Discussions on the Power Schedule Results

Table 6.6 shows a summary of the results on the power schedule costs, battery usage and the scalability of the algorithms described in this study.

In terms of the resultant global cost and the battery usage, A2C was the best of the four algorithms analyzed. While the differences in the grid power purchase costs for the algorithms were insignificant, the battery degradation cost differences were remarkable. This marked difference in the degradation cost which was a major cause of global cost differences implies that the whole power scheduling process may be viewed as a battery scheduling exercise. It can be deduced from these results that cost minimization in this scheduling exercise is best achieved through prudent battery usage. Under static optimization conditions, the best scheduling algorithm is the one that best uses the battery, i.e., by recommending shorter charge and discharge cycles. The battery degradation model presented in this study revealed that the battery degradation cost increases exponentially with the size of the charge/discharge cycles. Furthermore, the battery temperature increases with the charge/discharge power. A small difference in the battery energy profile can cause a very significant difference in the final global cost in the scheduling task. Therefore, shorter battery charge/discharge cycles are cheaper than long ones.

A2C was also found to have a better scalability than all the four algorithms. This is because it uses artificial neural networks to predict state/action values and performed fewer updates per episode than all the algorithms. Although the AC algorithm also used neural networks to predict state/action values, it performed over 12 times more updates per episode than the A2C algorithm. Thus, its computational speed and scalability were considerably low compared to the A2C algorithm. The asynchronous Q-learning algorithm was found to be the least scalable of all the four algorithms for two reasons. First, it employed a Q-table to track learning. When the state or action space is increased, the computational time increased considerably. Second, the restrictions introduced on its action space in each state to guide the agent in action selection led to redundancies in the Q-table. While the restrictions introduced led to a better performance in global cost and lower sensitivity to the learning rate than the conventional Q-learning method, these benefits came at the expense of scalability and speed. Although the conventional Q-learning algorithm also used

a Q-table to track learning, the absence of redundancies in the Q-table made it compute the solutions faster and more scalable than the asynchronous Q-learning.

Table 6.6: Summary of the Power Schedule Costs, Battery Energy Usage and Scalability of the Algorithms

Algorithm	Global Cost (USD)	Grid Power Purchase cost (USD)	Battery degradation cost (USD)	STD of Battery Energy Profile (kWh)	Scalability
A2C	155	147	8	16.0	8.51
Asynchronous Q-Learning	162	153	9	16.9	0.07
Conventional AC	179	151	28	20.9	0.14
Conventional Q-Learning	188	150	38	22.6	3.96

6.5.2 Algorithmic Limitations

Q-learning algorithms are simple and have better guarantee of convergence. However, the algorithms have some drawbacks. First, they have limited scalability due to the nature of the Q-table. An increase in the state-action space causes an increase in the size of the Q-table. This significantly limits the scalability of Q-table based methods. In comparison, the asynchronous Q-learning algorithm was found to have lower scalability than the conventional Q-learning algorithm due to redundancies in the Q-table. Nonetheless, the asynchronous Q-learning method produced more stable learning characteristics and returned a lower global cost. If computational resources are not a limiting factor, the asynchronous Q-learning would be preferred over the conventional Q-learning algorithm. Second, the convergence of the Q-learning methods is limited by the effectiveness of the strategy employed in action selection. In this study, a simple epsilon greedy method was used for action selection. Under different reward functions, the action selection method produces marked performance differences. The negative reward function was found to counteract the influence of the epsilon greedy algorithm. Also, it is quite challenging to tune the epsilon greedy algorithm to get the best balance between exploration and exploitation.

Actor-critic methods are more scalable because they do not need a Q-table to track learning. They use the weight matrices of neural networks to approximate the values of actions in each state.

Nonetheless, the conventional Q-learning was found to be more scalable than the conventional AC method. This may be because the environment was discretized, and the optimization was static. Q-table based Q-learning cannot be use for dynamic optimization in a continuous environment.

The main drawback of actor-critic algorithms is that they are difficult to tune. This is because more than one neural network model is involved. Although there is no action selection strategy needed to enable proper exploration in the actor-critic algorithms, the balance between bias and variance is a major challenge. If updates are done at every transition, like in the conventional AC algorithm, there is a risk of the model being biased and being trapped in a local optimum. If updates wait for too many transitions to be gathered before they are done, high variance results in the learning which may cause poor convergence. In the A2C algorithm, a balance is struck by employing experience replay. Instead of waiting for an episode to end before updating the neural networks, the updates are done in steps whose sizes are much less than the number of timesteps in a full episode. In terms of scalability, the A2C algorithm was found to be more scalable and trained faster than the conventional AC method. It was established that the number of updates per episode significantly affects the computational time and scalability of the actor-critic algorithms. The A2C algorithm trained faster than the conventional AC algorithm. This was because fewer updates were done per episode in A2C than in AC. For the same neural network model, scalability may be significantly improved by employing fewer calculated updates.

6.6 Summary

This chapter has presented the simulation data, the results and discussions for this study. Some important notable outcomes of the study include:

i) The proposed asynchronous Q-learning returned about 14% lower global cost than the conventional Q-learning.

ii) The limitation of action space in Q-learning algorithm improves the stability and the convergence but at the expense of computational speed and scalability.

iii) A negative reward function counterbalances the effect of the epsilon greedy action selection strategy in Q-learning resulting in poor convergence and instability.

iv) The proposed A2C algorithm returned about 13 % lower global cost than the conventional AC algorithm.

v) Experience replay and parallelization of learning agents in A2C introduces noise that reduces bias but increases variance in the learning process. The two techniques make A2C to be superior to the conventional AC algorithm in solving the power scheduling problem discussed in this study.

vi) A negative reward function returns better learning outcome in A2C than a positive reward function in the power scheduling problem described in this study.

vii) Battery degradation cost is a significant contributor to the global cost returned by the power scheduling algorithm in a grid-tied PV-battery system. The best scheduling algorithm for minimizing the energy cost in such a system is that which protects the battery from destructive charge/discharge oscillations.

7.0 Conclusions and Recommendations

7.1 Conclusions

The main objective of this dissertation was to develop and evaluate the performance of reinforcement learning techniques for optimal power scheduling in a grid-connected PV/battery electric vehicle charging station. Two reinforcement learning algorithms i.e., asynchronous Q-learning and A2C, were developed to minimize the energy cost in charging electric vehicles and compared with their conventional counterparts, i.e., the conventional Q-learning and AC algorithms correspondingly. From the results obtained in the study, it was evident that power scheduling in grid-connected microgrid with a battery energy storage boils down to a battery energy scheduling problem. This was realized from the way the battery utilization was a key factor that differentiated the global costs returned by the various algorithms explored in this research. Therefore, an accurate battery degradation model is key in improving the accuracy of the solutions produced by the power scheduling algorithms.

In the design of the MDP for the power management problem, battery energy scheduling may be viewed as an optimal path finding process, with the path being the battery energy profile. It was observed that designing reinforcement learning algorithms to solve the power scheduling problem is a complex balancing process. First, introducing more guides in the path of an agent in a Q-learning algorithm, i.e., restricting the allowable actions in each state, improves the stability with respect to the learning rate. It also leads to lower global cost. However, "micromanaging" the learning agent in this manner introduces redundancies in the design of the learning environment. Such redundancies require more computational resources to accommodate. Therefore, the designer needs to consider the system size and the computational resources available before introducing such action space restrictions. Otherwise, the algorithm's scalability would become a limiting factor. Second, it is not obvious that a reward function that intuitively achieves the objective of the learning process will return the desired learning trajectories and optimal solutions. In Q-learning, the way the Q-table has been initialized and the nature of the exploration strategy employed must be considered when choosing the reward function. It was observed that negative rewards counteract the influence of the epsilon-greedy algorithm if the Q-table is initialized with zeros. Thus, the learning trajectory becomes crooked and the final solutions are sub-optimal.

Third, although the use of artificial neural networks improves the scalability of the reinforcement learning algorithms, it was observed that it comes with a complex balancing to be done, i.e., the bias-variance balance. In the case of the actor-critic algorithms, performing updates in each transition as with the conventional actor-critic algorithm comes with a high bias. A high bias may limit exploration, thus, leading to poor convergence. Experience replay and parallelization of learning agents introduce some noise in the learning process. The noise injection reduces the bias as seen with the A2C algorithm. However, care must be taken on the amount of noise to be added in the algorithm because it may increase the variance in the learning profile to a point that may render the final solution inaccurate. Therefore, the designer of an actor-critic algorithm must limit the noise to just the amount that achieves the learning objective without affecting the stability of the algorithm.

7.2 Recommendations

The outcome of this research highlights an outstanding concern in reinforcement learning and the entire field of artificial intelligence, i.e., the issue of the complex tuning of algorithmic hyperparameters. It is a great limiting factor in the design of the reinforcement learning algorithms. Hand-coded tuning of the algorithms may limit the extent of learning they can achieve. If the reinforcement learning algorithms are to be trained to perform more complex tasks such as the power management in modern and future power systems, it would be important that the tuning of their core hyperparameters be learnt by the algorithms too. Adaptive tuning of some hyperparameters like learning rate and discount factor has been explored in the past. Nonetheless, some of the key hyperparameters such as number of nodes and hidden layers in a model, the size of the replay buffer in experience replay technique and the number of agents to be parallelized remain open to further investigation. Furthermore, conventional ways of reward engineering assume that the human designer understands the system dynamics. However, that is not the case with complex systems such as the modern and future power systems. As power systems become more complex, reward generation will need to be learnt as well. Future studies on the topic investigated in this dissertation should consider the following:

i) Adaptive tuning of hyperparameters such as the size of the replay buffer, the number of nodes in a neural network layer, etc.

ii) Implementation of a more intrinsic reward scheme that is generated by the agent itself as opposed to the conventional ones that are generated by the environment. This may help scale the reward functions to accommodate the convoluted dynamics in modern and future power systems.

iii) Investigation of the application of modern reinforcement learning real-time dynamic stochastic power scheduling. This would bridge the gap between the simulations and real-time environment.

iv) Exploration of the optimal scheduling and interoperability of interconnected charging stations under dynamic conditions.

References

[1] J. Larminie and J. Lowry, *Electric Vehicle Technology Explained*. Oxford: John Wiley and Sons Ltd, 2012.

[2] R. H. Schallenberg, “Prospects for the Electric Vehicle: A Historical Perspective,” *IEEE Transactions on Education*, vol. 23, no. 3, pp. 137–143, 1980.

[3] R. Garcia-Valle and J. A. P. Lopes, *Electric vehicle integration into modern power networks*. New York: Springer Science+Business Media, 2013.

[4] J. L. A. Jorge Martins, Francisco P. Brito, Delfim Pedrosa, Vítor Monteiro, *Real-Life Comparison Between Diesel and Electric Car Energy Consumption, in Grid Electrified Vehicles: Performance, Design and Environmental Impacts*. New York: Nova Science Publishers, 2013.

[5] M. O. Badaway, “Grid Tied PV/Battery System Architecture and Power Management For Fast EV Charging,” Ph.D Thesis, Department of Electrical Engineering, The University of Akron, 2016.

[6] Z. Sun, Z. Yang, Q. Niu, A. Foley, and K. Li, “Impact of Electric Vehicles on a Carbon Constrained Power System—A Post 2020 Case Study,” *Journal of Power and Energy Engineering.*, vol. 03, no. 04, pp. 114–122, 2015.

[7] X. Liu, D. Hildebrandt, and D. Glasser, “Environmental impacts of electric vehicles in South Africa,” *South African Journal of Science*, vol. 108, no. 1–2, 2012.

[8] U. Abronzini, C. Attaianese, M. D’Arpino, M. Di Monaco, A. Genovese, G. Pede and G. Tomasso, “Optimal energy control for smart charging infrastructures with ESS and REG,” in *2016 International Conference on Electrical Systems for Aircraft, Railway, Ship Propulsion and Road Vehicles and International Transportation Electrification Conference, ESARS-ITEC 2016*, 2016, pp. 1–6.

[9] H. Huo, Q. Zang, M. Q. Wang, D. G. Streets, and K. He, “Environmental Implication of Electric Vehicles in China,” *Environmental Science and Technology*, vol. 44, no. 13, pp. 4856–4861, 2010.

[10] S. Parhizi, H. Lotfi, A. Khodaei, and S. Bahramirad, “State of the art in research on microgrids: A review,” *IEEE Access*, vol. 3, pp. 890–925, 2015.

[11] E. A. Jasmin, “Reinforcement Learning Approaches to Power System Scheduling.” Ph.D Thesis, School of Engineering, Cochin University of Technology, 2008.

[12] S. Dong, “Grid Integration of Renewable Energy Sources via Virtual Synchronous Generator,” Ph.D Thesis, Department of Electrical and Computer Engineering, University of British Columbia, Vancouver, 2019.

[13] U. B. Tayab, F. Yang, M. El-Hendawi, and J. Lu, “Energy Management System for a Grid-Connected Microgrid with Photovoltaic and Battery Energy Storage System,” in *2018 Australian & New Zealand Control Conference (ANZCC)*, 2019, pp. 141–144.

[14] M. El-Hendawi, H. A. Gabbar, G. El-Saady, and E. N. A. Ibrahim, “Enhanced MG with optimum operational cost of pumping water distribution systems,” in *2017 5th IEEE International Conference on Smart Energy Grid Engineering, SEGE 2017*, 2017, pp. 137–142.

[15] G. K. Venayagamoorthy, R. K. Sharma, P. K. Gautam, A. Member, A. Ahmadi, and S. Member, “Dynamic Energy Management System for a Smart Microgrid,” *IEEE Trans. Neural Networks Learning Systems*, vol. 27, no. 8, pp. 1643–1656, 2016.

[16] B. V. Mbuwir, F. Ruelens, F. Spiessens, and G. Deconinck, “Battery energy management in a microgrid using batch reinforcement learning,” *Energies*, vol. 10, no. 11, pp. 1–19, 2017.

[17] M. O. Badawy and Y. Sozer, “Power Flow Management of a Grid Tied PV-Battery System for Electric Vehicles Charging,” *IEEE Transactions on Industry Applications*, vol. 53, no. 2, pp. 1347–1357, 2017.

[18] A. O. Erick and K. A. Folly, “Reinforcement Learning Approaches to Power Management in Grid-tied Microgrids : A Review,” in *Clemson University Power Systems Conference*, 2020, pp. 1–6.

[19] D. E. Olivares, A. Mehrizi-Sani, A. H. Etemadi, C. A. Cañizares, R. Iravani, M. Kazerani, A. H. Hajimiragha, O. Gomis-Bellmunt, M. Saeedifard, R. Palma-Behnke, G. A. Jiménez-

Estévez,and N. D. Hatziargyriou, “Trends in microgrid control,” *IEEE Transactions on Smart Grid*, vol. 5, no. 4, pp. 1905–1919, 2014.

[20] N. Bizon, N. Mahdavi Tabatabaei, and H. Shayeghi, *Analysis, Control and Optimal Operations in Hybrid Power Systems*. London: Springer Science+Business Media, 2014.

[21] A. O. Erick and K. A. Folly, “Energy Trading in Grid-connected PV-Battery Electric Vehicle Charging Station,” in *2020 International SAUPEC/RobMech/PRASA Conference, Cape Town, South Africa*, 2020, pp. 1–6.

[22] World Energy Council, “World Energy Resources 2016,” *World Energy Council*, pp. 6–46, 2016.

[23] World Economic Forum, “Electric Vehicles for Smarter Cities: The Future of Energy and Mobility,” *World Economic Forum*, January, 2018.

[24] M. Yilmaz and P. Krein, “Review of Charging Power Levels and Infrastructure for Plug-In Electric and Hybrid Vehicles and Commentary on Unidirectional Charging,” *IEEE Transactions on Power Electronics*, vol. 28, no. 5, pp. 2151–2169, 2012.

[25] O. Veneri, L. Ferraro, C. Capasso, and D. Iannuzzi, “Charging infrastructures for EV: Overview of technologies and issues,” in *Electrical Systems for Aircraft, Railway and Ship Propulsion, ESARS*, 2012, pp. 1–6.

[26] C. Youssef, E. Fatima, E. S. Najia, and A. Chakib, “A technological review on electric vehicle DC charging stations using photovoltaic sources,” in *IOP Conference Series: Materials Science and Engineering*, 2018, vol. 353, no. 1.

[27] X. Yang, L. Zhu, Z. Zhang, L. Li, and H. Wang, “Electric vehicles charging and discharging control strategy based on independent DC micro-grid,” *Proc. 2018 IEEE 3rd Adv. Inf. Technol. Electron. Autom. Control Conf. IAEAC 2018*, pp. 969–973, 2018.

[28] K. Qian, C. Zhou, M. Allan, and Y. Yuan, “Modeling of load demand due to EV battery charging in distribution systems,” *IEEE Transactions on Power Systems*, vol. 26, no. 2, pp. 802–810, 2011.

[29] C. Corchero, M. Cruz-Zambrano, F. J. Heredia, J. I. Cairo, L. Igualada-Gonzalez, and A. Romero-Ortega, “Optimal sizing of microgrids: A fast charging station case,” in *9th*

International Conference on the European Energy Market, EEM 12, 2012, pp. 1–6.

[30] M. Hijjo, F. Felgner, and G. Frey, “PV-Battery-Diesel microgrid layout design based on stochastic optimization,” in *2017 6th International Conference on Clean Electrical Power: Renewable Energy Resources Impact, ICCEP 2017*, 2017, pp. 30–35.

[31] U. Abronzini, C. Attaianese, M. D’Arpino, M. Di Monaco, A. Genovese, G. Pede and G. Tomasso, “Multi-source power converter system for EV charging station with integrated ESS,” in *2015 IEEE 1st International Forum on Research and Technologies for Society and Industry, RTSI 2015 - Proceedings*, 2015, no. 1, pp. 427–432.

[32] U. Abronzini, C. Attaianese, M. D’Arpino, M. D. Monaco, and G. Tomasso, “Power Converters for PV Systems with Energy Storage: Optimal Power Flow Control for EV’s Charging Infrastructures,” in *PCIM Europe 2016; International Exhibition and Conference for Power Electronics, Intelligent Motion, Renewable Energy and Energy Management*, 2016, no. May, pp. 1–7.

[33] B. V. Mbuwir, F. Spiessens, and G. Deconinck, “Self-learning agent for battery energy management in a residential microgrid,” in *Proceedings - 2018 IEEE PES Innovative Smart Grid Technologies Conference Europe, ISGT-Europe 2018*, 2018, pp. 1–6.

[34] Y. Ji, J. Wang, J. Xu, X. Fang, and H. Zhang, “Real-time energy management of a microgrid using deep reinforcement learning,” *Energies*, vol. 12, no. 12, 2019.

[35] A. Khamees, N. Badra, and A. Abdelaziz, “Optimal Power Flow Methods: A Comprehensive Survey,” *International Electrical Engineering Journal*, vol. 7, no. 4, pp. 2228–2239, 2016.

[36] S. S. Rao, *Engineering Optimization Theory and Practice*, 4th ed. Hoboken, New Jersey: John Wiley & Sons, Inc, 2009.

[37] S. D. Pendleton, H. Andersen, X. Du, X. Shen, M. Meghjani, Y. H. Eng, D. Rus and M. H. Ang Jr., “Perception, planning, control, and coordination for autonomous vehicles,” *Machines*, vol. 5, no. 1, pp. 1–54, 2017.

[38] C. J. Boo and H. C. Kim, “Photovoltaic based electric vehicle charging optimization,” *International Journal of Software Engineering and its Applications*, vol. 9, no. 12, pp. 285–

292, 2015.

[39] D. Zhang, “Optimal Design and Planning of Energy Microgrids.” Ph.D Thesis, Department of Chemical Engineering, University College London, 2013.

[40] I. P. S. on P. Engineering, *Modern Heuristic Oprimization Techniques Theory and Applications to Power Systems*. John Wiley & Sons, Inc, 2008.

[41] J. Zhu, *Optimization of Power Systems Operation*. John Wiley & Sons, Inc, 2009.

[42] A. Paffrath, M. Henneberger, and J. Mayer, “Particle Swarm Optimization: Basic Concepts, Variants and Applications in Power Systems,” *IEEE Transactions on Evolutionary Computing*, vol. 12, no. 2, 2008.

[43] M. O. Badawy and Y. Sozer, “Power flow management of a grid tied PV-battery powered fast electric vehicle charging station,” in *2015 IEEE Energy Conversion Congress and Exposition, ECCE 2015*, 2015, pp. 4959–4966.

[44] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*, 2nd ed. The MIT Press, 2018.

[45] V. François-lavet, D. Taralla, E. Damien, and R. Fonteneau, “Deep Reinforcement Learning Solutions for Energy Microgrids Management,” in *European Workshop on Reinforcement Learning*, 2016, pp. 1–7.

[46] Y. Wan, M. Zaheer, M. White, and R. S. Sutton, “Model-based Reinforcement Learning with Non-linear Expectation Models and Stochastic Environments,” in *FAIM Workshop on Prediction and Generative Modeling in Reinforcement Learning*, 2018, pp. 1–5.

[47] R. S. Sutton, “On the significance of Markov decision processes.” Department of Computer Science, University of Massachusetts, Armherst, MA USA, 2005.

[48] A. Potapov and M. K. Ali, “Convergence of reinforcement learning algorithms and acceleration of learning,” *Physical Review E - Statistical, Nonlinear, and Soft Matter Physics*, vol. 67, no. 2, pp. 267061–2670612, 2003.

[49] V. François-Lavet, “Contributions to deep reinforcement learning and its applications in smartgrids.” Ph.D Thesis, Department of Electrical Engineering and Computer Science,

University of Lie`ge, 2017.

[50] V. François-Lavet, P. Henderson, R. Islam, M. G. Bellemare and J. Pineau, "An Introduction to Deep Reinforcement Learning," *Found. trends Mach. Learn.*, vol. 11, no. 3–4, pp. 219–354, 2018.

[51] R. Bellman, "The Theory of Dynamic Programming." The Rand Corporation, California, pp. 1–550, 1954.

[52] Q. T. Tran, N. A. Luu, and T. L. Nguyen, "Optimal energy management strategies of microgrids," in *2016 IEEE Symposium Series on Computational Intelligence, SSCI 2016*, 2017, pp. 1–6.

[53] B. Jeddi, Y. Mishra, and G. Ledwich, "Dynamic programming based home energy management unit incorporating PVs and batteries," in *IEEE Power and Energy Society General Meeting*, 2018, pp. 1–5.

[54] R. Bellman, "A Markovian Decision Process," *Journal of Mathematics and Mechanics*, vol. 6, no. 5, pp. 679--684, 1957.

[55] C. J. C. H. Watkins, "Learning from Delayed Rewards," Ph.D Thesis, Department of Computer Science, University of Cambridge, 1989.

[56] E. Kuznetsova, Y. F. Li, C. Ruiz, E. Zio, G. Ault, and K. Bell, "Reinforcement learning for microgrid energy management," *Energy*, vol. 59, pp. 133–146, 2013.

[57] R. Leo, R. S. Milton, and S. Sibi, "Reinforcement learning for optimal energy management of a solar microgrid," in *2014 IEEE Global Humanitarian Technology Conference - South Asia Satellite, GHTC-SAS 2014*, 2014, pp. 183–188.

[58] E. Foruzan, L. K. Soh, and S. Asgarpoor, "Reinforcement Learning Approach for Optimal Distributed Energy Management in a Microgrid," *IEEE Transactions on Power Systems*, vol. 33, no. 5, pp. 5749–5758, 2018.

[59] Z. Yang, Y. Xie, and Z. Wang, "A Theoretical Analysis of Deep Q-Learning," 2019.

[60] R. S. Sutton, S. Singh, and D. McAllester, "Comparing Policy-Gradient Algorithms," *AT&T Shannon Lab. 180 Park Avenue, Florham Park. NJ 07932 USA*, 2000.

[61] A. W. M. Leslie P. Kaellbling, Michael L. Lttman, “Reinforcement Learning: A Survey,” *Journal of Artificial Intelligence Research*, vol. 4, pp. 237–285, 1996.

[62] B. J. Claessens, S. Vandael, F. Ruelens, K. De Craemer, and B. Beusen, “Peak shaving of a heterogeneous cluster of residential flexibility carriers using reinforcement learning,” in *4th IEEE/PES Innovative Smart Grid Technologies Europe, ISGT Europe 2013*, 2013, pp. 1–5.

[63] G. Shi, D. Liu, and Q. Wei, “Echo state network-based Q-learning method for optimal battery control of offices combined with renewable energy,” *IET Control Theory and Applications*, vol. 11, no. 7, pp. 915–922, 2017.

[64] J. R. Vázquez-Canteli and Z. Nagy, “Reinforcement learning for demand response: A review of algorithms and modeling techniques,” *Applied Energy*, vol. 235, pp. 1072–1089, 2019.

[65] S. Lange, T. Gabel, and M. Riedmiller, “Batch reinforcement learning,” *Adaptation, Learning, and Optimization*, vol. 12, pp. 45–73, 2012.

[66] D. Silver, G. Lever, N. Heess, T. Degris, D. Wierstra, and M. Riedmiller, “Deterministic policy gradient algorithms,” *JMLR W&CP*, vol. 32, pp. 605–619, 2014.

[67] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg and D. Hassabis, “Human-level control through deep reinforcement learning,” *Nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[68] A. Nair, P. Srinivasan, S. Blackwell, C. Alcicek, R. Fearon, A. De Maria, V. Panneershelvam, M. Suleyman, C. Beattie, S. Petersen, S. Legg, V. Mnih, K. Kavukcuoglu and D. Silver, “Massively Parallel Methods for Deep Reinforcement Learning,” *Google DeepMind, London*. pp. 1–14, 2015.

[69] X. Lu, X. Xiao, L. Xiao, C. Dai, M. Peng, and H. V. Poor, “Reinforcement Learning-Based Microgrid Energy Trading With a Reduced Power Plant Schedule,” *IEEE Internet Things Journal*, vol. 6, no. 6, pp. 10728–10737, 2019.

[70] Z. Zhang, D. Zhang, and R. C. Qiu, “Deep Reinforcement Learning for Power System: An Overview,” *CSEE J. Power Energy Syst.*, vol. 6, no. 1, pp. 213–225, 2020.

[71] J. Wang, K. Zeb, T. Dhruva, S. Hubert, L. Joel, M. Remi, C. Blundell, K. Dharshan, B. Matt, “Learning to reinforcement learn,” *DeepMind, London, UK*. pp. 1–17, 2016.

[72] D. Fuselli, F. De Angelis, M. Boaro, S. Squartini, Q. Wei, D. Liu and F. Piazza, “Action dependent heuristic dynamic programming for home energy resource scheduling,” *International Journal of Electrical Power and Energy Systems*, vol. 48, no. 1, pp. 148–160, 2013.

[73] Y. Wei, Z. Zhang, F. R. Yu, and Z. Han, “Power allocation in HetNets with hybrid energy supply using actor-critic reinforcement learning,” in *2017 IEEE Global Communications Conference, GLOBECOM 2017 - Proceedings*, 2017, pp. 1–5.

[74] Z. Wan, H. Li, and H. He, “Residential Energy Management with Deep Reinforcement Learning,” in *Proceedings of the International Joint Conference on Neural Networks*, 2018, pp. 1–7.

[75] H. Hua, Y. Qin, C. Hao, and J. Cao, “Optimal energy management strategies for energy Internet via deep reinforcement learning approach,” *Applied Energy*, vol. 239, no. 1, pp. 598–609, 2019.

[76] T. Hirata, D. B. Malla, K. Sakamoto, Y. Okada, and T. Sogabe, “Smart Grid Optimization by Deep Reinforcement Learning over Discrete and Continuous Action Space,” *Bulletin of Networking, Computing, Systems, and Software*, vol. 8, no. 1, pp. 19–22, 2019.

[77] P. Chen, M. Liu, C. Chen, and X. Shang, “A battery management strategy in microgrid for personalized customer requirements,” *Energy*, vol. 189, no. C, pp. 1–18, 2019.

[78] P. Odonkor and K. Lewis, “Control of Shared Energy Storage Assets Within Building Clusters Using Reinforcement Learning,” in *Proceedings of the ASME 2018 International Design Engineering Technical Conferences & Computers and Information in Engineering Conference*, 2018, pp. 1–11.

[79] L. Yu, W. Xie, D. Xie, Y. Zou, D. Zhang, Z. Sun, L. Zhang, Y. Zhang and T. Jiang, “Deep Reinforcement Learning for Smart Home Energy Management,” *IEEE Internet Things*

Journal, vol. 7, no. 4, pp. 2327–4662, 2019.

[80] E. Mocanu, D. C. Mocanu, P. H. Nguyen, Member, A. Liotta, M. E. Webber, M. Gibescu, and J.G. Slootweg, "On-Line Building Energy Optimization Using Deep Reinforcement Learning," *IEEE Transactions on Smart Grid*, vol. 10, no. 4, pp. 3698–3708, 2019.

[81] R. Haji Mahdizadeh Zargar and M. H. Yaghmaee Moghaddam, "Development of a Markov-Chain-Based Solar Generation Model for Smart Micro-grid Energy Management System," *IEEE Transactions on Sustainable Energy*, vol. 11, no. 2, pp. 736–745, 2020.

[82] D. Dewey, "Reinforcement Learning and the Reward Engineering Principle Rewards in an Uncertain World," in *AAAI Spring Symposium Series*, 2014, pp. 1–8.

[83] C. J. C. H. Watkins and P. Dayan, "Technical Note: Q-Learning," *Machine Learning*, vol. 8, no. 3, pp. 279–292, 1992.

[84] M. A. L. Thathachar and P. S. Sastry, *Networks of learning automata : Techniques for Online Stochastic Optimization*. New York: Springer Science+Business Media, 2004.

[85] Y. Xi, L. Chang, M. Mao, P. Jin, N. Hatziargyriou, and H. Xu, "Q-learning algorithm based multi-agent coordinated control method for microgrids," in *9th International Conference on Power Electronics-ECCE Asia*, 2015, pp. 1497–1504.

[86] E. A. Jasmin, T. P. Imthias Ahamed, and V. P. Jagathy Raj, "Reinforcement learning solution for unit commitment problem through pursuit method," in *2009 International Conference on Advances in Computing, Control, and Telecommunication Technologies*, 2009, pp. 324–327.

[87] M. Tokic, "Adaptive ε-greedy exploration in reinforcement learning based on value differences." Institute of Neural Information Processing, University of Ulm, 89069 Ulm, Germany, 2010.

[88] S. Haykin, *Neural Networks and Learning Machines: A Comprehensive Foundation*. New Jersey: Pearson Prentice Hall, 2008.

[89] A. Shrestha and A. Mahmood, "Review of deep learning algorithms and architectures," *IEEE Access*, vol. 7, pp. 53040–53065, 2019.

[90] M. R. Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, “Playing Atari with Deep Reinforcement Learning,” *Deepmind Technologies*, pp. 1–9, 2013.

[91] R. Lincoln, S. Galloway, B. Stephen, and G. Burt, “Comparing Policy Gradient and Value Function Based Reinforcement Learning Methods in Simulated Electrical Power Trade,” *IEEE Transactions on Power Systems*, vol. 27, no. 1, pp. 373–380, 2012.

[92] Y. P. Awate, “Policy-gradient based actor-critic algorithms,” in *Proceedings of the 2009 WRI Global Congress on Intelligent Systems, GCIS 2009*, 2009, vol. 3, pp. 505–509.

[93] V. Veeriah, H. Van Seijen, and R. S. Sutton, “Forward actor-critic for nonlinear function approximation in reinforcement learning,” in *Proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems, AAMAS*, 2017, vol. 1, pp. 556–564.

[94] K. Arulkumaran, M. P. Deinseroth, M. Brundage, and A. A. Bharath, “Deep Reinforcement Learning: A Brief Survey,” *IEEE Signal Processing Magazine*, vol. 17, no. 8, pp. 26–38, 2017.

[95] S. Bai, Y. Du, and S. Lukic, “Optimum design of an EV/PHEV charging station with DC bus and storage system,” in *2010 IEEE Energy Conversion Congress and Exposition, ECCE 2010 - Proceedings*, 2010, pp. 1178–1184.

[96] “IEEE Standard Technical Specifications of a DC Quick Charger for Use with Electric Vehicles,” *IEEE Std 2030.1.1-2015*, pp. 1–97, 2015.

[97] R. Xiong, J. Cao, Q. Yu, H. He, and F. Sun, “Critical Review on the Battery State of Charge Estimation Methods for Electric Vehicles,” *IEEE Access*, vol. 6, pp. 1832–1843, 2017.

[98] C. J. M. R. Catarina, P. Tiago, F. Pedro, R. Sergio, V. Zita, B. Jose, S. Jono, N. Maria, “Dynamic electricity tariff definition based on market price, consumption and renewable generation patterns,” in *IEEE 2018 Clemson University Power Systems Conference (PSC)*, 2019, pp. 1–5.

[99] Y. Wang, R. Wang, and H. Zhang, “Research of Dynamic Time-of-use Tariffs for PEV Charging Stations Based on User Behavior Habits,” in *2nd IEEE Conference on Energy*

Internet and Energy System Integration, EI2 2018 - Proceedings, 2018, pp. 1–5.

[100] S. K. Hoke Anderson, Brissette Alexander and P. Annabelle, “Accounting for Lithium-Ion Battery Degradation in Electric Vehicle Charging Optimization,” *IEEE Journal of Emerging and Selected Topics in Power Electronics,* vol. 2, no. 3, pp. 691–700, 2014.

[101] J. McDowal, “Understanding Lithium-Ion Technology,” in *Proceedings of Battcon*, 2008, pp. 1–10.

[102] K. J. Laidler, “The development of the arrhenius equation,” *Journal of Chemical Education.*, vol. 61, no. 6, pp. 494–498, 1984.

[103] B. Xu, “Degradation-limiting Optimization of Battery Energy Storage Systems Operation,” Master’s Thesis*,* Power Systems Laboratory, Federal Institute of Technology Zurich, D¨attwill, Switzerland, 2013.

[104] C. Zhou, K. Qian, M. Allan, and W. Zhou, “Modeling of the Cost of EV Battery Wear Due to V2G Application in Power Systems,” *IEEE Transactions on Energy Conversion.*, vol. 26, no. 4, pp. 1041–1050, 2011.

[105] V.-H. Bui, A. Hussain, and H.-M. Kim, “Double Deep Q-Learning-Based Distributed Operation of Battery Energy Storage System Considering Uncertainties,” *IEEE Transactions on Smart Grid*, vol. 11, no. 1, pp. 457–469, 2019.

[106] T. Tongloy, S. Chuwongin, K. Jaksukam, C. Chousangsuntorn, and S. Boonsang, “Asynchronous deep reinforcement learning for the mobile robot navigation with supervised auxiliary tasks,” in *2017 2nd International Conference on Robotics and Automation Engineering, ICRAE 2017*, 2017, pp. 68–72.

[107] OpenAI, “OpenAI Baselines ACKTR $ A2C,” 2020. [Online]. Available: https://openai.com/blog/baselines-acktr-a2c/. [Accessed: 05-Apr-2020].

[108] Q. Sun, C. Du, Y. Duan, H. Ren, and H. Li, “Design and application of adaptive PID controller based on asynchronous advantage actor–critic learning method,” *Wireless Networks*, 2019.

[109] T. Lehtola and A. Zahedi, “Cost of EV Battery Wear due to Vehicle to Grid Application,” in *2015 Australasian Universities Power Engineering Conference (AUPEC), Wollongong,*

NSW, pp. 1–4, 2015.

[110] K. K. Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Tim Harley, Timothy P. Lillicrap, David Silver, "Asynchronous Methods for Deep Reinforcement Learning," in *JMLR: W&CP, Proceedings of the 33rd International Conference on Machine Learning, New York, USA*, 2016, vol. 48, pp. 1–11.

[111] E. Nikishin, P. Izmailov, B. Athiwaratkun, D. Podoprikhin, T. Garipov, P. Shvechikov, D. Vetrov, A. G. Wilson, "Improving Stability in Deep Reinforcement Learning with Weight Averaging," in *UAI workshop on Uncertainty in Deep Learning*, 2018, pp. 1–5.

[112] Y. Wu, E. Mansimov, S. Liao, R. Grosse, and J. Ba, "Scalable trust-region method for deep reinforcement learning using Kronecker-factored approximation," *Advances in Neural Information Processing Systems*, vol. December, pp. 5280–5289, 2017.

Appendices

Appendix A: The Policy Gradient Theorem and the A2C Algorithm

A.1: Proof of Policy Gradient Theorem

A proof of convergence of policy gradient as a gradient ascent problem is provided here. The proof has been adapted from the work by Sutton *et al*. [60].

With regards to the advantage function, the policy gradient theorem means that the gradient of the agent's objective may be defined as the expected value of the advantage multiplied by the gradient of the log of the current policy. That is,

$$\nabla_\theta J(\theta) = \mathbb{E}[A(s,a)\nabla_\theta log\pi(a|s)] \tag{1}$$

Where $J(\theta)$ is the objective of the learning agent, $\pi(a|s)$ is the prevailing policy and the advantage is defined as:

$$A(s,a) = Q(s,a) - V(s) \tag{2}$$

To prove (1) above, we estimate the expectation value of the term in square brackets. Expectation is the mean of an ensemble, therefore, (1) can be re-expressed as:

$$\nabla_\theta J(\theta) \approx \frac{1}{N}\sum_{i=1}^{N}[A(s_i,a_i)\nabla_\theta log\pi(a_i|s_i)] \tag{3}$$

Since the agent is maximizing the total rewards and the rewards are not related to the neural network parameter vector θ, we sample the rewards under the policy π, defined by the neural network parameters. Given a sequence of transitions to a terminal state, T, say, $\mathbb{Z} = \{(s_1,a_1),(s_2,a_2),(s_3,a_3),\ldots,(s_T,a_T)\}$, we can define the total returns as:

$$R(\mathbb{Z}) = \sum_{t=1}^{T}[r(s_t,a_t)] \tag{4}$$

The objective, J may be defined in terms of the neural network parameter vector as:

$$J(\theta) = \mathbb{E}[R(\mathbb{Z});\ \pi_\theta] = \sum_{\mathbb{Z}}[P(\mathbb{Z}:\theta)R(\mathbb{Z})] \tag{5}$$

Thus,

$$\nabla_\theta J(\theta) = \sum_{\mathbb{Z}} \nabla_\theta [P(\mathbb{Z}:\theta)R(\mathbb{Z})] \tag{6}$$

Or, multiplying by 1 in terms of $P(\mathbb{Z}:\theta)/\ P(\mathbb{Z}:\theta)$,

$$\nabla_\theta J(\theta) = \sum_{\mathbb{Z}} \frac{P(\mathbb{Z}:\theta)}{P(\mathbb{Z}:\theta)} \nabla_\theta [P(\mathbb{Z}:\theta)R(\mathbb{Z})] \tag{7}$$

Which can be reorganized into:

$$\nabla_\theta J(\theta) = \sum_{\mathbb{Z}} P(\mathbb{Z}:\theta) \frac{\nabla_\theta [P(\mathbb{Z}:\theta)]}{P(\mathbb{Z}:\theta)} R(\mathbb{Z}) \tag{8}$$

By chain rule, for a given differentiable function *f*,

$$\nabla log f \ = \frac{\nabla f}{f} \tag{9}$$

It follows that, (6) can be written as:

$$\nabla_\theta J(\theta) = \sum_{\mathbb{Z}} [P(\mathbb{Z}:\theta) \nabla_\theta log P(\mathbb{Z}:\theta) R(\mathbb{Z})] \tag{10}$$

Since the expectation operator implies the sum of the values weighted by the probability distribution over the ensemble. Therefore, (10) may be expressed as:

$$\nabla_\theta J(\theta) = \mathbb{E}[\nabla_\theta log P(\mathbb{Z}:\theta) R(\mathbb{Z})] \tag{11}$$

Equation (11) above means that the gradient of the agent's objective may be expressed in terms of the parameters of the neural network represented by the vector θ. Therefore, gradient ascent can be applied to maximize the objective. The term in square bracket has the gradient of the log of a product. It can therefore be expressed as the gradient of the sum of the logs of each terms in the product as:

$$\nabla_\theta J(\theta) = \mathbb{E}[\nabla_\theta \{log P(\mathbb{Z}:\theta) + log R(\mathbb{Z})\}] \tag{12}$$

Since $log R(\mathbb{Z})$ does not depend on the neural network parameter vector θ, its gradient with respect to θ collapses to zero. Equation (12) then becomes:

$$\nabla_\theta J(\theta) = \mathbb{E}[\nabla_\theta log P(\mathbb{Z}: \theta)] \tag{13}$$

If the problem is Markovian, then we define the transition probability for the episode $\mathbb{Z}$ as:

$$P(\mathbb{Z}: \theta) = \sum_{t=1}^{T} P(s_{t+1}|s_t, a_t)\pi_\theta(a_t|s_t) \tag{14}$$

Noting that Equation (13) expresses the gradient of the objective as the expectation value of the gradient of the log of $P(\mathbb{Z}: \theta)$, we express the log of the right-hand side of (14) as the sum of log of the two variables. Since in (14), $P(s_{t+1}|s_t, a_t)$ does not depend on the parameters of the neural network, the gradient of its log collapses to zero, thus:

$$\nabla_\theta J(\theta) = \mathbb{E}\left[\sum_{t=1}^{T}\{\nabla_\theta log\pi_\theta(a_t|s_t)R(\mathbb{Z})\}\right] \tag{15}$$

In (15), instead of estimating the total returns of an episode, we substitute it with the advantage function and have:

$$\nabla_\theta J(\theta) = \mathbb{E}\left[\sum_{t=1}^{T}\{\nabla_\theta log\pi_\theta(a_t|s_t)A(s_t, a_t)\}\right] \tag{16}$$

If we apply the chain rule to Equation (16), we get back to Equation (1). Thus, the proof of the policy gradient theorem.

A.2: Impact of Number of Nodes in the Hidden Layer

The neural network models for both the actor and the critic networks presented in this study each have 1 hidden layer. In Fig. A.1, a plot of the learning curves for A2C obtained using different number of nodes in the hidden layer from 50 to 300 in steps of 50. Also, the learning curve obtained using the chosen value of 256 blue (see the blue curve) was included.

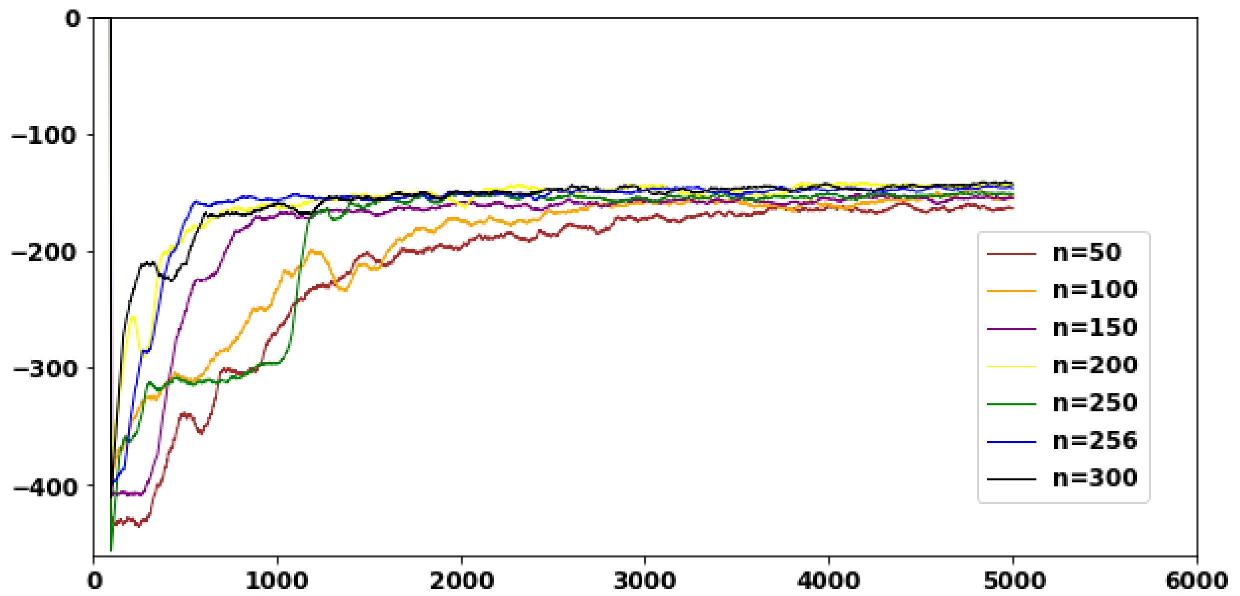

Fig. A: Learning curves for A2C using different number of nodes on the hidden layer of the actor and the critic models

Appendix B: Load, PV, and Grid Tariff Data

Table B: Input Data

Time	Load Profile (kW)	PV Profile (kW)	Grid Tariff Profile (USD/kWh)
0	5	0	0.22
1	5	0	0.21
2	0	0	0.21
3	10	0	0.21
4	20	0	0.35
5	40	0	0.4
6	58	7	0.45
7	80	20	0.35

8	76	37	0.39
9	43	50	0.38
10	22	70	0.37
11	60	60	0.2
12	41	43	0.15
13	22	37	0.39
14	10	32	0.39
15	21	25	0.38
16	46	15	0.38
17	56	7	0.37
18	65	1	0.18
19	80	0	0.19
20	50	0	0.18
21	18	0	0.28
22	15	0	0.27
23	10	0	0.27

Appendix C: Simulation Tools

Table C: Python tools for machine learning

Python Tool	Function
Numpy	Manipulation of arrays
Pandas	Reading of excel files
Pytorch	Implementation of neural networks.
Matplotlib	Plotting graphs and visualizing learning behaviors
Random	Random number generation

All the algorithms were implemented on a 64-bit Windows 10 Intel (R) Core i7 CPU, 1.80GHz with 8GB RAM.

Appendix D: Python Codes

Python implementation of the algorithms and the corresponding environment are provided here below.

D.1: Learning Environment Code

```
import pandas as pd
import random
import numpy as np

def read_state_var(k):

    data_file = 'trading_data.xlsx'
    data = pd.read_excel(data_file)
    return(list(data.loc[k]))

init_state_vectors = [read_state_var(k) for k in range(24)]

class EvChargingEnvironment():

    def __init__(self, init_state_vectors):

        self.init_state_vectors = init_state_vectors
```

```
    def get_state_dictionary(self, state_vectors):

        return {i[0]:tuple([i[1], i[2], i[3]]) for i in state_vectors}


    def get_initial_state(self, time, states, Eb_0 = 50):

        init_state = list(states[float(time)])
        init_state.append(Eb_0)
        state = init_state

        return state

    def next_state(self, action, states, time):


        state = list(states[time])
        state.append(action)


        return state

    def calculate_cost(self, state, action, Eb_max = 100, delta_t = 1, C_bt
=400):

        if action == 0:
            action = action + 10
        else:
            action = action * 10
        doD_1 = 1 - (state[3])/Eb_max
        doD_2 = 1 - (action)/Eb_max

        l_doD_1 = 694*(doD_1)**(-0.795)
        l_doD_2 = 694*(doD_2)**(-0.795)
        cost_bd_DoD = C_bt*abs((1/l_doD_2) - (1/l_doD_1))

        SoC_avg1 = (2 - (state[3] + action)/Eb_max)
        cost_bd_T = 0.00125*abs(state[3]-action) + 0.02
        cost_bd_SoC_avg1 = 0.05*SoC_avg1 - 0.019

        p_bat = state[3] - action
        cost_bd = abs(max(cost_bd_T, cost_bd_SoC_avg1,
cost_bd_DoD)*p_bat*delta_t)
        p_grid = state[0] - state[1] - p_bat

        if p_grid < 0:

            cost_grid = 0

        else:
            cost_grid = p_grid*state[2]

        cost = cost_bd + cost_grid
```

```
        return cost

    def calculate_reward(self, cost):

        return -1*cost

        #return 1/(cost+1)**2


    def timing(self, time):
        return float(time+1)

    def done(self, time):

        return True if time == 23 else False
```

D.2: Conventional Q-learning Code

```
states = [read_state_var(k) for k in range(24)]
actions = [i for i in range (10, 100,10)]
q = [[0 for x in actions] for y in states]

def take_action(actions, states, state, epsilon):

    global action

    greedy =
actions[(q[states.index(state)]).index(max(q[states.index(state)]))]

    if random.uniform(0, 1) > (epsilon):

        action = greedy

    else:
         action = np.random.choice(actions)


    return action

def calculate_cost (state, action, Ebk_1, Eb_max = 100, delta_t = 1, C_bt =
400):

    global cost

    doD_1 = 1 - (Ebk_1)/Eb_max
    doD_2 = 1 - (action)/Eb_max

    l_doD_1 = 694*(doD_1)**(-0.795)
    l_doD_2 = 694*(doD_2)**(-0.795)
    cost_bd_DoD = C_bt*abs((1/l_doD_2) - (1/l_doD_1))

    SoC_avg1 = (2 - (Ebk_1 + action)/Eb_max)
```

```
    cost_bd_T1 = 0.00125*abs(Ebk_1-action) + 0.02
    cost_bd_SoC_avg1 = 0.05*SoC_avg1 - 0.019

    p_bat = Ebk_1 - action
    cost_bd = abs(max(cost_bd_T1, cost_bd_SoC_avg1,
cost_bd_DoD)*p_bat*delta_t)
    p_grid = state[1] - state[2] - p_bat

    if p_grid < 0:
        cost_grid = 0

    else:

        cost_grid = p_grid*state[3]

    cost = cost_bd + cost_grid


    return cost

def calculate_reward(cost):


    return -1*cost

    return 1/(cost+1)**2

def learning_1(alpha= 0.001, gamma = 1.0):

    global action, cost, reward

    n = 0
    epsilon = 1
    min_rate = 0.1
    max_rate = 1.0
    decay = 0.00325
    cost_list = []
    reward_list = []

    while n <= 15000:
        k = 0

        cost_episode = 0
        episode_reward = 0
        state = states[k]
        Ebk_1 = 50

        for k in range(24):

            action = take_action(actions,states, state, epsilon)

            cost = calculate_cost(state, action,Ebk_1)
            cost_episode += cost
```

```
            k += 1

            state = states[k]

            reward = calculate_reward(cost)
            episode_reward += reward
            if k == 23:
                q[states.index(states[k-1])][actions.index(action)] += \
                alpha*(reward - q[states.index(states[k-
1])][actions.index(action)])
                break

            else:
                q[states.index(states[k-1])][actions.index(action)] += \
                alpha*(reward + gamma*(max(q[states.index(states[k])]) - \
                q[states.index(states[k-1])][actions.index(action)]))

            Ebk_1 = action

        n += 1
        cost_list.append(cost_episode)
        reward_list.append(episode_reward)

        epsilon = min_rate + (max_rate - min_rate)*np.exp(-decay * n)



    return q, reward_list, cost_list
```

D.3: Asynchronous Q-learning Code

```
import pandas as pd
import random
import numpy as np
import matplotlib as plt

q = {}
costs = []
reward_list = []

def read_state_var(k):
    data_file = 'trading_data.xlsx'
    data = pd.read_excel(data_file)
    return(list(data.loc[k]))

state_vectors = [read_state_var(k) for k in range(24)]

def possible_actions(state_id, delta_t = 1, Eb_max = 100, Eb_min = 10):

    state_vector = [float(i) for i in (state_id.split('*'))]
    pb_k_min = int((state_vector[4] - Eb_max)/delta_t)

    pb_k_max = int((state_vector[4] - Eb_min)/delta_t)
```

```
    delta_p_k = state_vector[1] - state_vector[2]
    possible_actions = [[pg_k, pb_k] for pg_k in range(-50, 90)
                        for pb_k in range(pb_k_min, pb_k_max,10) if pg_k +
pb_k == delta_p_k]

    actions_dict = {}
    actions = ['*'.join(str(x) for x in i) for i in possible_actions]
    for i in actions:
        actions_dict[i] = 0.0

    return actions_dict, possible_actions


def take_action(state_id_current, epsilon):

    actions = q[state_id_current]
    greedy =
list(actions.keys())[list(actions.values()).index(max([q[state_id_current][u]
for u in q[state_id_current]]))]

    if random.uniform(0, 1) > (epsilon):

        action = greedy

    else:
         action = np.random.choice(list(set(list(actions.keys()))))

    return action


def next_state(state_id_current, action, k, delta_t = 1):

    state_vector = [float(i) for i in (state_id_current.split('*'))]
    action_vector = [float(i) for i in (action.split('*'))]
    if k <= 23:
        state_data = state_vectors[k]
        for i in range(len(state_data)):
            state_vector[i]=state_data[i]


    else:
        state_vector[0] = 24.0

    state_vector[4] -= (action_vector[1])*delta_t
    return state_vector, action_vector

def power_schedule_cost(action, state_id_current, Eb_max = 100, delta_t = 1,
C_bt=400):

    global l_doD_1, l_doD_2

    state_vector = [float(i) for i in (state_id_current.split('*'))]
    action_vector = [float(i) for i in (action.split('*'))
```

```
    doD_1 = 1- state_vector[4]/Eb_max
    doD_2 = 1 - (state_vector[4] + action_vector[1])/Eb_max
    if doD_1 ==0:
        cost_bd_DoD = 0

    elif doD_2 == 0:
        cost_bd_DoD = 0

    else:
        l_doD_1 = 694*(doD_1)**(-0.795)
        l_doD_2 = 694*(doD_2)**(-0.795)
        cost_bd_DoD = C_bt*abs((1/l_doD_2) - (1/l_doD_1))

    SoC_avg = (2*state_vector[4] + action_vector[1]*delta_t)/(2*Eb_max)
    cost_bd_T = 0.00125*abs(action_vector[1]) + 0.02
    cost_bd_SoC_avg = 0.05*SoC_avg - 0.019
    cost_bd = abs(max(cost_bd_T, cost_bd_SoC_avg,
cost_bd_DoD))*abs(action_vector[1])*delta_t

    if action_vector[0] >= 0:

        cost_g = (action_vector[0])*abs(state_vector[3])*delta_t
    else:
        cost_g = 0

    return cost_bd+cost_g, cost_bd, cost_g

def calculate_reward(cost_bd, cost_g):

    return -1*(cost_bd + cost_g)
    #return 1/(cost+1)**2


#Main learning loop
def learning(a = 0.001, y = 1.0):

    n = 0
    epsilon = 1
    min_rate = 0.1
    max_rate = 1
    decay = 0.00325

    while n <= 15000:
        k = 0
        epi_cost = 0
        x = state_vectors[0]
        state_vector = x
        actions = []
        Eb_k = 50
        epi_reward = 0
        state_vector.append(Eb_k)

        for k in range(24):
```

```
            state_id_current = '*'.join([str(x) for x in state_vector])

            actions_dict_current, possible_actions_current =
possible_actions(state_id_current)
            if state_id_current not in q:
                q[state_id_current] = actions_dict_current

            action = take_action(state_id_current, epsilon)

            cost, cost_bd, cost_g = power_schedule_cost(action,
state_id_current)


            epi_cost += cost


            k += 1
            state_vector, action_vector = next_state(state_id_current,
action, k)

            reward = calculate_reward(cost_bd, cost_g)
            epi_reward += reward

            state_id_next= '*'.join([str(x) for x in state_vector])

            actions_dict_next, possible_actions_next =
possible_actions(state_id_next)
            if state_id_next not in q:
                q[state_id_next] = actions_dict_next

            if k == 23:
                q[state_id_current][action] += a*(reward-
q[state_id_current][action])
                break

            else:
                q[state_id_current][action] += a*(reward +
y*((max([q[state_id_next][u] for u in q[state_id_next]]))) -
q[state_id_current][action])

        costs.append(epi_cost)
        reward_list.append(epi_reward)
        state_vectors[0].pop()
        n += 1
        epsilon = min_rate + (max_rate - min_rate)*np.exp(-decay * n)

    return costs, reward_list
```

D.4: Conventional Actor-critic Code

```
import math
import random
import numpy as np
import matplotlib
```

```python
import matplotlib.pyplot as plt
from collections import namedtuple
from itertools import count
import torch as T
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F

class GenericNetwork(nn.Module):
    def __init__(self, lr, input_dims, fc1_dims, n_actions):
        super(GenericNetwork, self).__init__()
        self.input_dims = input_dims
        self.lr = lr
        self.fc1_dims = fc1_dims

        self.n_actions = n_actions
        self.fc1 = nn.Linear(self.input_dims, self.fc1_dims)
        self.fc2 = nn.Linear(self.fc1_dims, n_actions)
        self.optimizer = optim.Adam(self.parameters(), lr = lr)

        self.device = T.device("cuda" if T.cuda.is_available() else "cpu")
        self.to(self.device)
        self.n_actions = len([i for i in range(10,100,10)])

    def forward(self, observations):
        state = T.tensor(observations).to(self.device)
        x = F.relu(self.fc1(state))

        x = self.fc2(x)

        return x

class Agent(object):

    def __init__(self, alpha, beta, input_dims, gamma = 1.0, l1_size = 256,
                 n_actions = len([i for i in range(10, 100, 10)])):

        self.gamma = gamma
        self.log_probs = None
        self.actor = GenericNetwork(alpha, input_dims, l1_size, n_actions)
        self.critic = GenericNetwork(beta, input_dims, l1_size, n_actions)

    def choose_action(self, observation):

        probabilities = F.softmax(self.actor.forward(observation))
        action_probs = T.distributions.Categorical(probabilities)
        action = action_probs.sample()
        self.log_probs = action_probs.log_prob(action)

        return action.item()

    def learn(self, state, reward, next_state, done):

        self.actor.optimizer.zero_grad()
```

```
        self.critic.optimizer.zero_grad()
        actor_value = self.actor.forward(state)
        critic_value = self.critic.forward(next_state)

        delta = ((reward + self.gamma*critic_value*(1-int(done))) -
critic_value)
        actor_loss = -1*self.log_probs*delta
        critic_loss = delta**2

        (actor_loss + critic_loss).sum().backward()

        self.actor.optimizer.step()
        self.critic.optimizer.step()


agent = Agent(alpha = 0.0001, beta = 0.0003, input_dims = 4, gamma = 1.0,
             l1_size = 256, n_actions = len([i for i in range(10, 100, 10)]))

env = EvChargingEnvironment(init_state_vectors)
score_history = []
costs = []
n_episodes = 5000
def run():
    for i in range (n_episodes):

        states = env.get_state_dictionary(init_state_vectors)
        time = 0.0
        done = False
        state = T.tensor(env.get_initial_state(time, states),
dtype=T.float32)

        epi_costs = 0.0


        while not done:

            action = agent.choose_action(state)

            cost = env.calculate_cost(state, action)
            epi_costs += cost

            reward = T.tensor(env.calculate_reward(cost, action),
dtype=T.float32)

            time = env.timing(time)
            next_state =T.tensor(env.next_state(action, states, time),
dtype=T.float32)
            done = env.done(time)
            agent.learn(state, reward, next_state, done)

            state = next_state

        costs.append(epi_costs)
    return costs, score_history
```

D.5: Advantage Actor-Critic Code

```
import torch
import sys
import numpy as np
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F
from torch.autograd import Variable
import matplotlib.pyplot as plt
import pandas as pd

hidden_size = 256
learning_rate = 0.0001
num_actions = len([i for i in range(10, 100, 10)])

GAMMA = 1.0
num_steps = 10
max_episodes = 5000

class ActorCritic(nn.Module):
    def __init__(self, num_inputs, num_actions, hidden_size,
learning_rate=3e-3):
        super(ActorCritic, self).__init__()

        self.num_actions = num_actions
        self.critic_linear1 = nn.Linear(num_inputs, hidden_size)
        self.critic_linear2 = nn.Linear(hidden_size, 1)

        self.actor_linear1 = nn.Linear(num_inputs, hidden_size)
        self.actor_linear2 = nn.Linear(hidden_size, num_actions)

    def forward(self, state):
        state = Variable(torch.from_numpy(state).float().unsqueeze(0))
        value = F.relu(self.critic_linear1(state))
        value = self.critic_linear2(value)

        policy_dist = F.relu(self.actor_linear1(state))
        policy_dist = F.softmax(self.actor_linear2(policy_dist), dim=1)

        return value, policy_dist


def a2c(env):
    num_inputs = 4
    num_outputs = len([i for i in range(10, 100, 10)])

    actor_critic = ActorCritic(num_inputs, num_outputs, hidden_size)
    ac_optimizer = optim.Adam(actor_critic.parameters(), lr=learning_rate)

    all_lengths = []
    average_lengths = []
```

```
    all_rewards = []
    entropy_term = 0
    costs = []

    for episode in range(max_episodes):
        log_probs = []
        values = []
        rewards_list = []
        rewards = 0
        cost_ep =0
        states = env.get_state_dictionary(init_state_vectors)
        time = 0.0
        done = False
        state = np.array(env.get_initial_state(time, states))

        for steps in range(num_steps):
            value, policy_dist = actor_critic.forward(state)
            value = value.detach().numpy()[0,0]
            dist = policy_dist.detach().numpy()

            action = np.random.choice(num_outputs, p=np.squeeze(dist))

            log_prob = torch.log(policy_dist.squeeze(0)[action])
            entropy = -np.sum(np.mean(dist) * np.log(dist))

            cost = env.calculate_cost(state, action)
            cost_ep += cost
            reward = env.calculate_reward(cost)
            rewards += reward

            time = env.timing(time)
            new_state =np.array(env.next_state(action, states, time))
            done = env.done(time)
            rewards_list.append(reward)
            values.append(value)
            log_probs.append(log_prob)
            entropy_term += entropy
            state = new_state
            if done:

                costs.append(cost_ep)

            if done or steps == num_steps-1:
                Qval, _ = actor_critic.forward(new_state)
                Qval = Qval.detach().numpy()[0,0]
                all_rewards.append(rewards)
                all_lengths.append(steps)
                average_lengths.append(np.mean(all_lengths[-10:]))
                if episode % 500 == 0:
                    sys.stdout.write("episode: {}, rewards: {}, total length:
{}, average length: {} \n".format(episode, np.sum(rewards), steps,
average_lengths[-1]))
                break
```

```
    Qvals = np.zeros_like(values)
    for t in reversed(range(len(rewards_list))):
        Qval = rewards_list[t] + GAMMA * Qval
        Qvals[t] = Qval

    values = torch.FloatTensor(values)
    Qvals = torch.FloatTensor(Qvals)
    log_probs = torch.stack(log_probs)

    advantage = Qvals - values
    actor_loss = (-log_probs * advantage).mean()
    critic_loss = 0.5 * advantage.pow(2).mean()
    ac_loss = actor_loss + critic_loss - 0.01 * entropy_term
    #ac_loss = actor_loss + critic_loss
    ac_optimizer.zero_grad()
    ac_loss.backward()
    ac_optimizer.step()
return all_rewards, costs
```