Automatic generation control of a two-area power system using deep reinforcement learning

S. Reynolds (BMa., BFin.)

This thesis is presented as part of the requirements for the conferral of the degree:

BACHELOR OF ENGINEERING HONOURS (ELECTRICAL)

Supervisors:
Dr. C. Yeo & Dr. S. Klaric

Charles Darwin University
College of Engineering

September 23, 2020

Declaration

I, *S. Reynolds (BMa., BFin.)*, declare that this thesis is submitted in partial fulfilment of the requirements for the conferral of the degree *BACHELOR OF ENGINEERING HONOURS (ELECTRICAL)*, from Charles Darwin University, is wholly my own work unless otherwise referenced or acknowledged. This document has not been submitted for qualifications at any other academic institution.

S. Reynolds (BMa., BFin.)

September 23, 2020

Abstract

Power systems behave in a non-linear fashion; however, traditional control architectures for maintaining power system frequency are designed assuming that plant can be modelled using ordinary linear differential equations. This assumption is reasonable for minor frequency deviations given the present level of non-linearity in power systems. Owing to an increase in the proportion of photovoltaic power generation, along with an increase in the use of battery energy storage systems, Australian power system dynamics are becoming increasingly non-linear. This is creating a need to explore novel control architectures to improve frequency control performance.

Reinforcement learning (RL) can be used to design controllers. Through trial and error, the RL agent learns what actions it must take to achieve a control objective. It does this with no prior knowledge of the system it is trying to control. This makes RL ideal when designing controllers for complex non-linear systems that are difficult to mathematically model. In the past, RL has been successfully applied to problems with low dimensional discrete state and action spaces. In the last five years, the integration of deep neural networks as function approximators in RL architectures has demonstrated impressive performance for control problems with high dimensional continuous state and action spaces. This approach is known as deep reinforcement learning (DRL), and is considered state-of-the-art for robotic applications.

This work applies a Deep Deterministic Policy Gradient (DDPG) DRL algorithm as a controller for the system frequency and transmission line power flow for a two-area power system. Each area is comprised of a single generator and a single load that is representative of aggregated generation and aggregated load demand for the respective areas. A single transmission line connects the two areas. Load demand for each power area fluctuates as consumers switch appliances on and off. If left unchecked this causes deviations of the system frequency away from 50Hz. Using a simulated power system model and load demand data sourced from a utility provider, the DDPG agent will be trained to control a regulating generator. Its control action can arrest frequency deviations, and restore each area to the scheduled value, under normal operating conditions.

The performance of the DDPG controller has been evaluated against classical control approaches to provide an assessment of the viability of DRL controllers for power system control.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In 2018, approximately 261TW h of power was generated in the Australian electricity sector. Renewables contributed 19% of the total generation, an increase from 15% in 2017. The Department of Industry, Science, Energy and Resources have observed an increase in renewable energy generation year-on-year in the electricity generation market since 2008, as shown in Figure 1.1 [1].

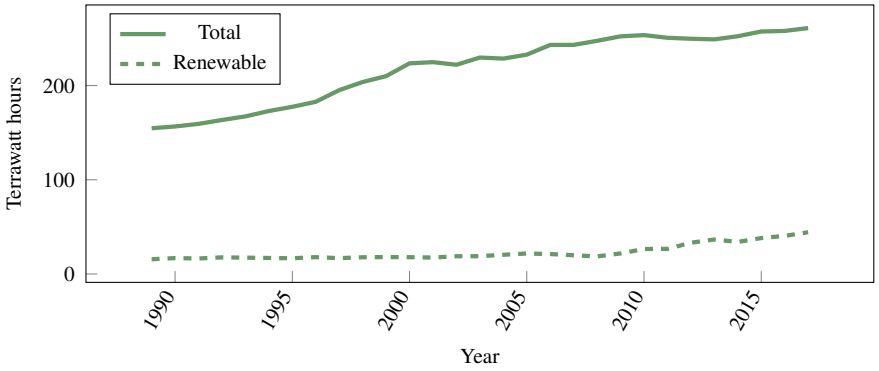

Figure 1.1: Power generation from renewable sources (dashed line), and total power generation (solid line) in Australia from 1977 to 2018.

One of the benefits of transitioning from thermal sources of power generation to renewable sources is reduced greenhouse gas emissions [2]; however, this transition is not without its drawbacks. With an increased reliance on renewable power generation sources posing challenges for power system stability owing to load management. A recent example is the system failure in Alice Springs, caused by an event cascade that was triggered by cloud cover shadowing a solar array. The system failure left residents in Alice Springs without power for approximately eight hours [3]. An independent investigation highlighted that poor control policies were one of the factors that contributed to the blackout. In this instance, the generator provisioned to ramp up in the event of cloud cover was unable to be controlled. Moreover, generators that were still under the control regime were issued operating set points above their rated capacity, which resulted in thermal overload

and subsequent tripping from the protection system [4].

Current control methods use classical feedback loop techniques. These methods can be brittle when faced with system changes, or scenarios which they were not designed for. An improved controller would be one that can learn and adapt its control protocols to an unseen system or event, given some broad control objective. This research proposes a deep reinforcement learning (DRL) agent for controlling the frequency of a power system consisting of multiple generators, and multiple load demands with individual stochastic profiles.

1.1 Research Aim

The principle aim of this research is to compare the performance of known, optimised feedback loop controller architectures against a DRL based control system when tasked with performing load following ancillary services with regulating generators under automatic generation control (AGC) for a two-area power system. This research will be undertaken in order to understand the feasibility of using DRL agents for two-area power system management.

1.2 Structure of Interim Report

The remainder of this report is structured as follows:

Chapter 2 introduces the necessary background to understand work presented later in the report. This includes a formal introduction to reinforcement learning, deep neural networks, and deep reinforcement learning, including associated mathematical preliminaries.

Chapter 3 undertakes a literature review exploring different technologies used to address the load frequency control problem. Topics discussed include feedback loops, fuzzy logic, genetic algorithms, and artificial neural networks. The chapter concludes with a review of DRL applications to load frequency control and the motivation for using these controllers.

Chapter 4 outlines the research approach, providing a discussion on model development and implementation, a framework for preliminary investigations, and concludes by establishing the main experimental approach for assessing DRL controller feasibility for load frequency control.

Chapter 5 presents results from a partially completed preliminary investigation.

Chapter 6 provides a discussion on findings from Chapter 5 and evaluates feasibility of the proposed DRL controller. Chapter concludes with recommendations for the continued direction of this work.

Chapter 2

Background

2.1 Power Systems and Frequency

Interconnected power systems are comprised of power generating units and energy storage systems connected to transmission and distribution networks. Generated power is used to service load demand. A single line diagram of a power network can be seen in Figure 2.1. The diagram shows how thermal generation units (left-hand side), such as coal and nuclear, in addition to renewable sources of generation, like wind and solar provide a power generation mix that is transmitted by a network for the consumers of generated energy: industry and households (right-hand side).

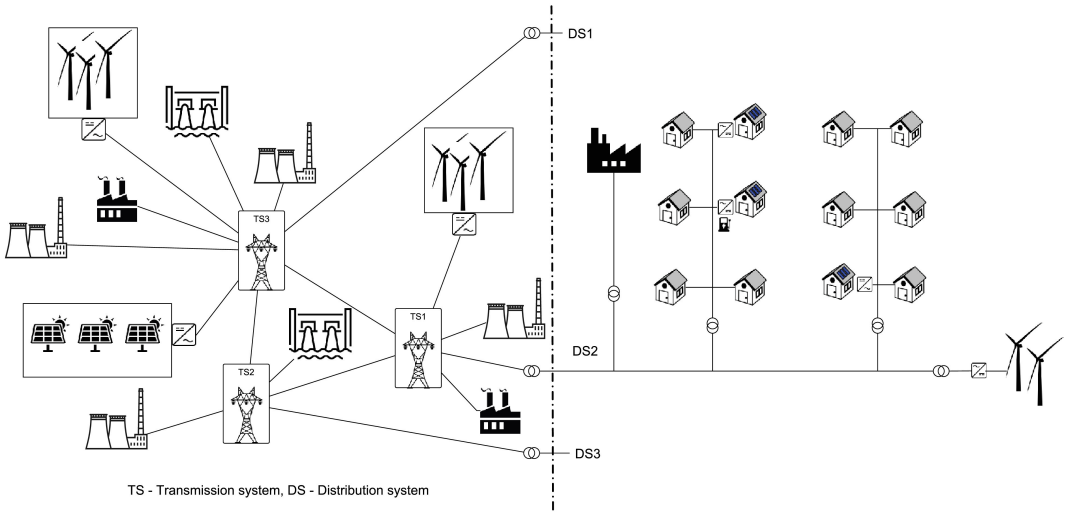

Figure 2.1: A single line diagram of a typical power system. The image shows points of generation from thermal and renewable sources, and the subsequent supply of generated energy to meet load demand through the transmission and distribution network [5].

One of the key elements to successful operation of interconnected power systems is ensuring total load demand is matched with total generation, while taking into account power losses involved with generation, transmission, and distribution [6]. To understand

why it is important to match generation with load demand consider the basic operation of a single thermal generator.

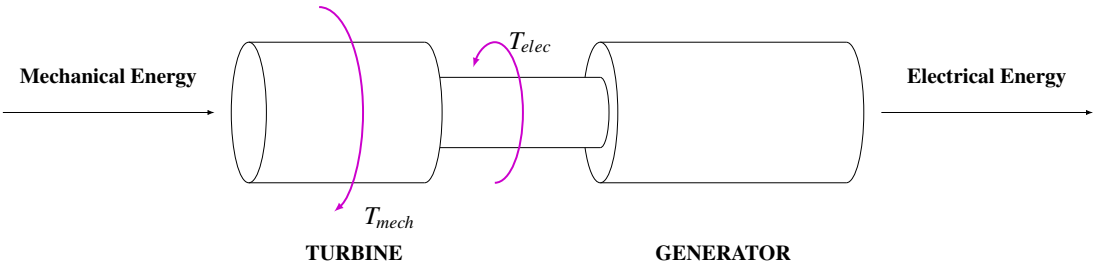

Figure 2.2: A thermal generation unit consisting of a prime mover (turbine), and a synchronous machine [6].

The essential elements of a thermal generator are a prime mover (such as a gas turbine) and a synchronous machine, as depicted in Figure 2.2. The prime mover provides mechanical torque, T_{mech}, which drives the synchronous machine producing electrical energy. In response, the synchronous machine creates an opposing torque that depends on the size of the load demand. This opposing torque is referred to as electrical torque and is denoted as T_{elec}. If α represents angular acceleration of the generator rotating mass, and I is its moment of inertia, then by Newton's second law:

$$T_{mech} - T_{elec} = I\alpha. \tag{2.1}$$

When T_{mech} equals T_{elec} the system will be in a steady state of zero angular acceleration with a constant angular velocity, ω. Now, if $T_{mech} > T_{elec}$, then the system has an angular acceleration causing the angular velocity, ω, to increase. This results in a frequency increase in the system. Conversely, if $T_{mech} < T_{elec}$ then the angular velocity ω will decrease, resulting in a frequency decrease. It is important to note that at any point in time, the total electrical load demand will fluctuate as consumers switch grid connected appliances or motors on and off. The result is that an uncontrolled system will have a continually changing frequency. Australia's electricity network is designed to operate at a frequency of 50Hz. In the majority of network scenarios, the Australian Energy Market Operator (AEMO) has a desired operating range for frequency which lies between 49.85 and 50.15Hz [7]. Similarly, the Power and Water Corporation (PWC) network technical code for the Northern Territory states that under normal operating conditions frequency should be maintained in the range of 49.80 to 50.20Hz [8]. Network operation outside of the specified range can cause damage to electrical equipment such as transformers or motors, which are designed to operate at specific frequencies [9]. Network designers engineer protection schemes so that sustained frequency excursions outside of the allowed range will cause equipment to trip from the network [10].

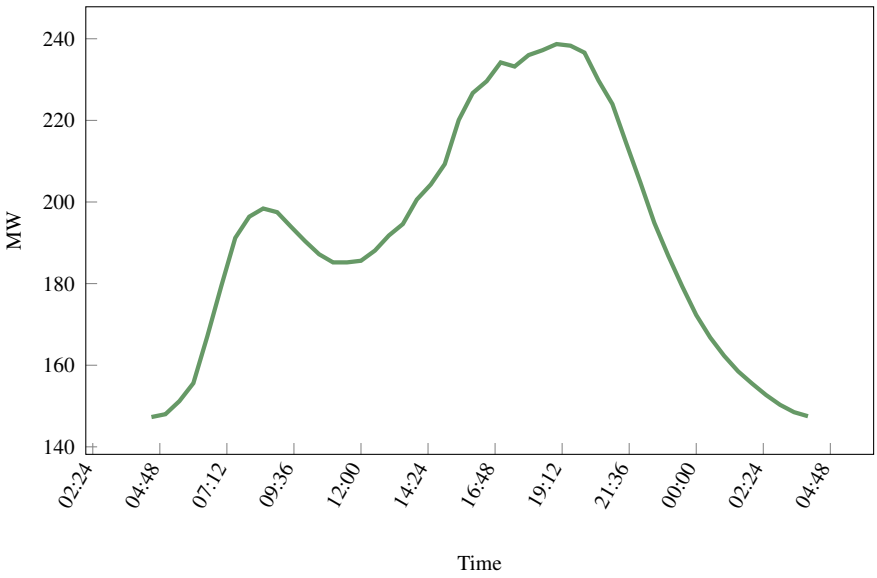

Figure 2.3: Weekday energy demand profile in South Australia during summer [11].

Protection schemes tripping equipment from the network is undesirable as this can leave consumers without power, resulting in economic loss. Further, if disconnections are uncontrolled the system stability is reduced [10]. System controllers, such as the AEMO and PWC, are interested in being able to control the system to follow changes in load demand so that system frequency is maintained in the allowable range. Additionally, they are interested in control mechanisms to restore frequency excursions as a result of unexpected disturbances. System controllers can use historical data, like that shown in Figure 2.3, to forecast daily demand profiles with some reliability. This type of forecasting does not help when trying to predict the occurrence of random disturbances; however, it does provide a starting point for estimating required generation needed to meet demand. It is important to note that forecasting is not perfect. Inevitably mismatches in supply and demand will occur causing small imbalances between T_{mech} and T_{elec}, resulting in a change to angular velocity ω and the network frequency [12]. To perfectly match supply and demand, system controllers use generators referred to as regulating units, placed under Automatic Generation Control (AGC) [13]. A regulating unit is a generator that has the capacity to increase or decrease mechanical torque T_{mech}, and the AGC is a system providing control over the mechanical torque output of regulating generators. If the system controller has a sufficient number of regulating units under AGC it can perform two functions: load following, and restoring the system to stable operating conditions in the event of a disturbance [14]. Using a regulating unit under AGC control to load follow is referred to, by AEMO, as load following ancillary services [15].

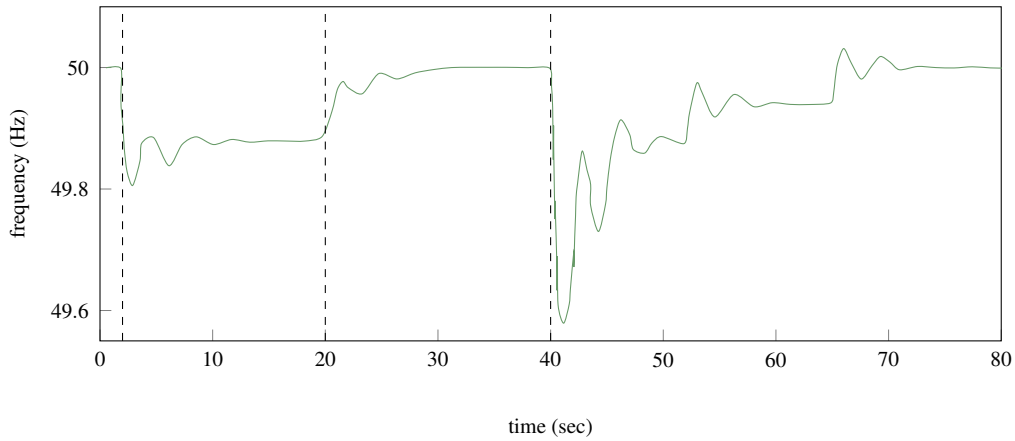

Figure 2.4: A minor frequency disturbance occurs at the 2 sec mark and primary control systems (governors) arrest the frequency drop. System frequency is adjusted to desired 50Hz operating level using AGC control of regulating units. This is referred to as supplementary (or secondary) control. At the 40 sec mark the network experiences a major frequency disturbance which is arrested by emergency control measures such as under-frequency load shedding (UFLS). System restoration is aided using AGC control of regulating units, which AEMO refers to as spinning reserve ancillary services [16].

Load following control adjusts regulating units in order to match supply with a demand load profile, and maintain frequency in a normal operating range a shown in the first 40 seconds of Figure 2.4. Using a regulator under AGC control to restore the system after a major disturbance is referred to, by AEMO, as providing spinning reserve ancillary services. [15]. When used in either fashion it is important to note that the regulating unit is not responsible for arresting frequency excursions, rather, it is used to restore the system back to the allowable frequency operating range after the frequency excursion has been arrested. An example of a frequency excursion, arrest, and subsequent restoration for minor and major disturbances can be seen in Figure 2.4. AEMO and PWC do not require all generators on the network to act as regulating units since adequate frequency control can be achieved using a subset of the total available generators.

2.1.1 Modelling a Single Area System

The power system model shown in Figure 2.1 depicts total generation coming from many generation assets. This is complex to model. Generation assets are often divided into subgroups referred to as control areas [13]. A control area is defined as a subset of generators that are in close proximity to each other and constitute a coherent group that speed up and slow down together, maintaining their relative power angles [13]. Therefore, the total network is comprised of many interconnected control areas. An example of this can be seen in Figure 2.5. Notice that for each area there is only one load and one generator.

Typically, for each control area, many loads will be aggregated into a single load, and many generators into a single generator, in order to simplify the model [14]. The simplest

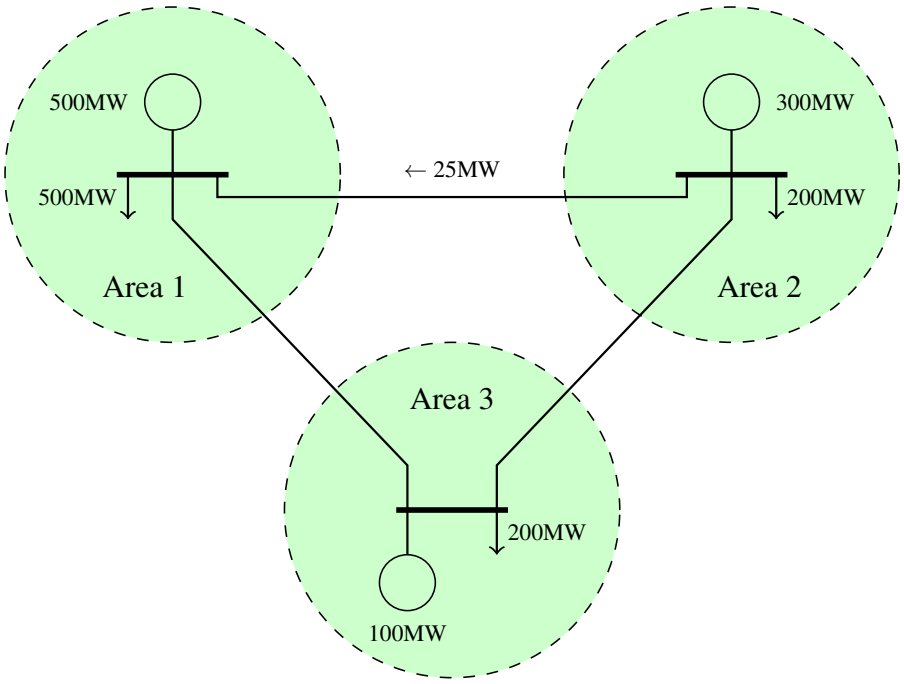

Figure 2.5: An example of three interconnected control areas in a 60Hz power system. The interconnections allow power to flow from one area to another, allowing generators to service loads from different areas. Each control area consists of several generators and loads, but are modelled with a single generator and single load for simplicity [14].

power system to control is one that consists of a single control area. A single control area power system has no interconnections to any other control area. It is comprised of a consumer load demand, and a set of generators, some of which are acting as regulating units. As previously mentioned, for modelling simplicity, loads are aggregated to a single load, and generators can be aggregated to a single generator. This simple system, shown in Figure 2.6 is well understood.

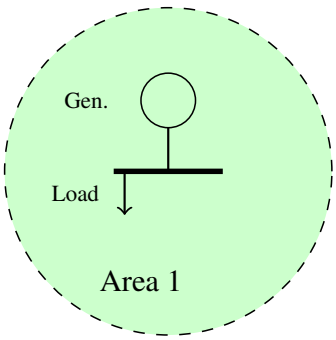

Figure 2.6: A single area power system has a load and a generator, with no connections to other power areas

When expressing the single area model mathematically, researchers consider three separate elements: the governor, the turbine, and the generator-load.

Governor Model

The governor is a device which is used to regulate the speed of a generator. It is often modelled as a first order system, as shown in Figure 2.7. The governor receives an input, $\Delta P_C(s)$, which represents a power increase/decrease command brought about by adjusting the governor speed changer. Older governors operated by mechanically coupling the governor with the rotating shaft of the generator; however, newer governors operate electronically. Owing to the older mechanical governor designs, governor models often include a proportional feedback loop using turbine rotational frequency change, $\Delta F(s)$. The output, $\Delta Y_E(s)$, represents a control signal to change the turbine steam valve position. The model parameter K_{sg} is the static gain of the speed-governing mechanism, and the parameter T_{sg} is the time constant of the speed-governing mechanism.

The frequency domain derivation of the governor model can be found in Appendix A.1.

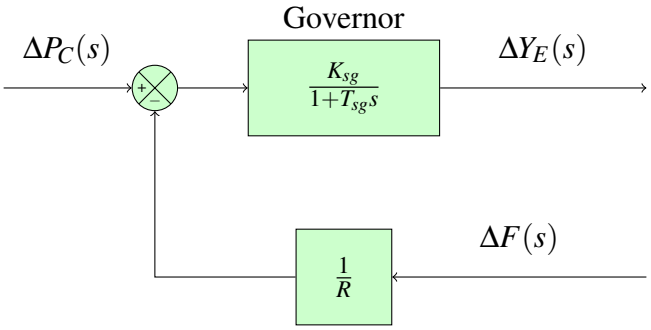

Figure 2.7: Frequency domain model for the generator governor.

Turbine Model

A steam turbine converts energy stored in the form of high pressure and high temperature steam into rotational energy. The steam is created in a boiler heated by fossil fuel combustion. Alternatively, a nuclear reactor could be used to produce the heat. As the steam is forced through turbine nozzle sections it is accelerated to high velocities. Kinetic energy from the high velocity steam is converted into shaft torque as it collides with moving blades attached to the turbine rotor. Coupling the turbine rotor to a synchronous machine converts rotational energy to electrical energy.

Steam turbines can come in a variety of configurations with multiple turbines, and steam reheating systems being common features in modern machinery. The model presented in this section has omitted many of these details, providing a simpler representation. The model, shown in Figure 2.8, is a first-order system that takes the input $\Delta Y_E(s)$, representing the change in steam valve position. The model output is denoted as $\Delta P_G(s)$, which represents the change in power output of the turbine. An increase in the $\Delta Y_E(s)$ lets additional steam into the turbine, increasing $\Delta P_G(s)$. Conversely, a decrease in $\Delta Y_E()$, restricts the steam inflow, reducing $\Delta P_G(s)$.

The frequency domain derivation of the governor model can be found in Appendix A.2.

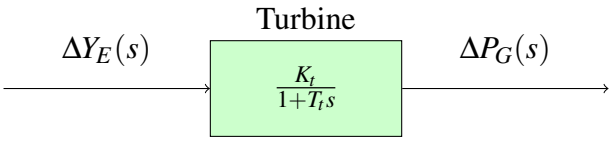

Figure 2.8: Frequency domain model for the generator turbine.

Generator Load Model

Consumers connected to the power system change their demand for power as appliances are switched on and off, placing demand on the generator. The incremental power demanded by consumers, $\Delta P_L(s)$, is subtracted from the incremental power delivered by the turbine to form the input for the generator-load model. The model, shown in Figure 2.9, is first-order and change in frequency as the output, denoted by $\Delta F(s)$. As the incremental power demanded increases, the generator frequency decreases. Conversely, as the incremental power demanded decreases, the generator frequency increases.

The frequency domain derivation of the governor model can be found in Appendix A.3.

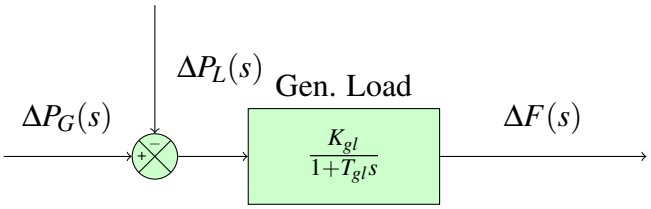

Figure 2.9: Frequency domain model for the generator-load.

2.1.2 Frequency Control for a Single Area System

It is common for a speed droop governor feedback control regime to perform primary frequency control, with an AGC feedback loop used to perform secondary frequency control when restoring a minor frequency excursion [6], [13], [14], [17]. A particularly well laid out approach to developing linear models for the turbine, generator, load, and governor was presented by Kundur [17]. The full model is shown in Figure 2.10. This model is described by a single regulating generator supplying a load. The governor block is a first-order linear model of the governor. The turbine block is a first-order linear model of the turbine. The final block is the generator-load, which is also a first-order linear model. The AGC feedback loop uses a proportional integral controller.

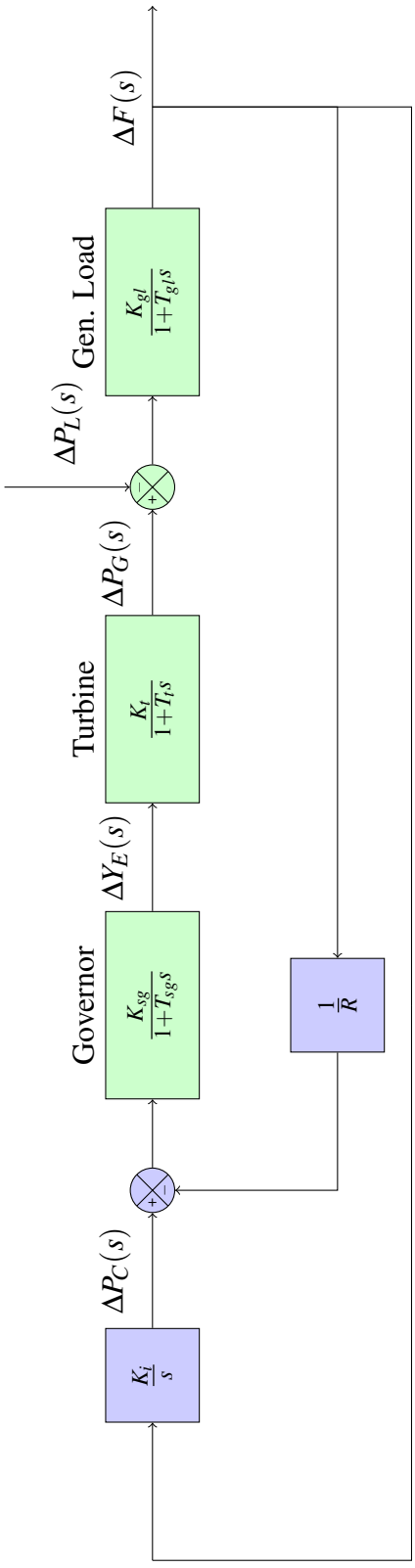

Figure 2.10: A classical feedback control approach for a single control area power system. The system is comprised of a first order models for both turbines, and generators. The governor controllers are also first order models. AGC is implemented using an integral control block in a feedback loop [17].

2.1.3 Modelling a Two Area System

The single area system presented in Section **??** is useful to help understand the role of governors and AGC in controlling power system frequency. In reality, power systems are comprised of many control areas connected by transmission lines, referred to as tie lines. Often, there is some net power transfer over the tie lines, dictated by contractual agreement. Single area models do not describe this additional complexity. The simplest model that includes tie lines is the two area power system, shown in Figure 2.11.

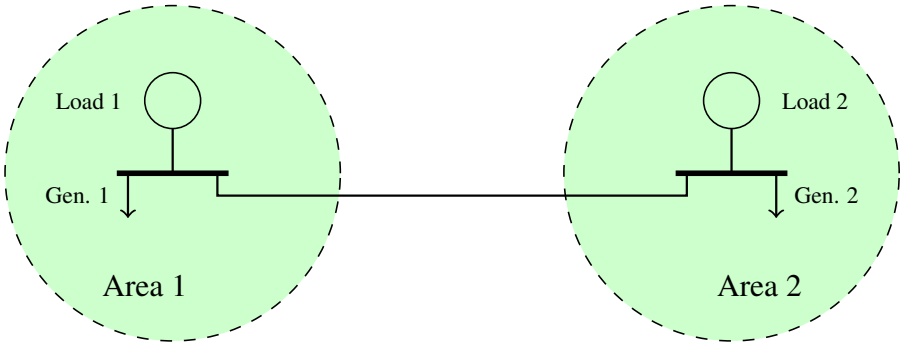

Figure 2.11: A two area power system comprised of generators and load connected via a tie line. Power flows from one area to the other depending on the power demands.

Governor, and turbine models presented in §**??** are used in each of the power areas for a two-area power system model. Additionally, a model for the tie-line connection between the two power areas is used, as well as an updated generator-load model.

Tie Line Model

The transmission line connecting two power areas exports power from one area to another, allowing an increase or decrease in the input to the generator-load model for each area. This effect is implemented using a feedback loop. Inputs to the tie-line model are derived from frequency outputs from both areas. The values are differenced and converted to a change in power by multiplying by 2π and integrating the result. Outputs, $\Delta P_{tie,1}$ and $\Delta_{tie,2}$, represent incremental power supplied or received over the tie-line. The tie-line model is shown in Figure 2.12. The model derivation can be found in Appendix B.1.

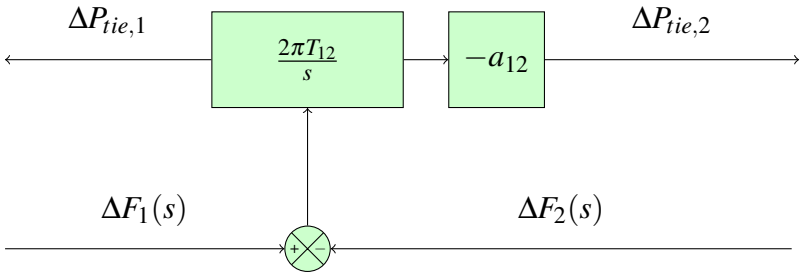

Figure 2.12: Frequency domain model for the transmission line between two power areas.

Generator Load Model for Two-Area Power System

As mentioned in §2.1.3, the generator-load model either supplies or receives power via the tie-line connection between the two areas. To accommodate for this a feedback loop is implemented, which adds tie-line power back into the generator-load, as shown in Figure 2.13.

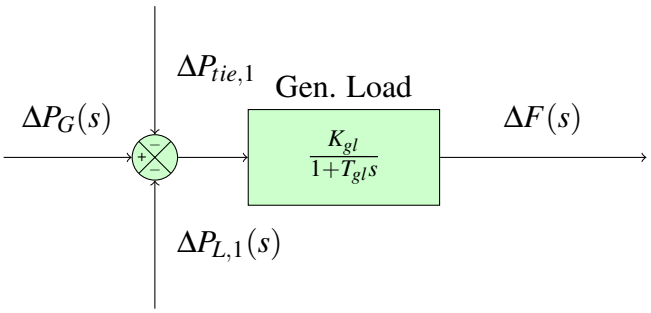

Figure 2.13: Frequency domain model for the generator-load with feedback from a tie-line.

The frequency domain derivation of the generator-load model for a two area power system can be found in Appendix B.2.

2.1.4 Frequency Control for Two Area System

The control objective for a two area power system is to maintain the inter-area power transfer, whilst regulating the frequency of each area. An AGC integral feedback loop on regulating units, like that shown in Figure 2.10, would ensure that power system frequency is maintained, however, would not guarantee inter-area power transfer agreements are observed.

Violation of power transfer contracts due to control issues does not allow for a stable market in which energy can be reliably traded. Fortunately, multi control area power systems are well understood, and classical control approaches have been successfully implemented to meet the new control objectives. In order to apply classical PI controllers to two-area power system control, a metric called Area Control Error (ACE) is used in the AGC feedback loop for each control area. ACE is a linear combination of the frequency deviations and tie-line power flows.

The two-area power system can be modelled using governor, turbine, tie-line and generator-load models developed in §**??** and §2.1.4. This is shown shown in Figure 2.14, in addition to the classical PI controller with ACE input.

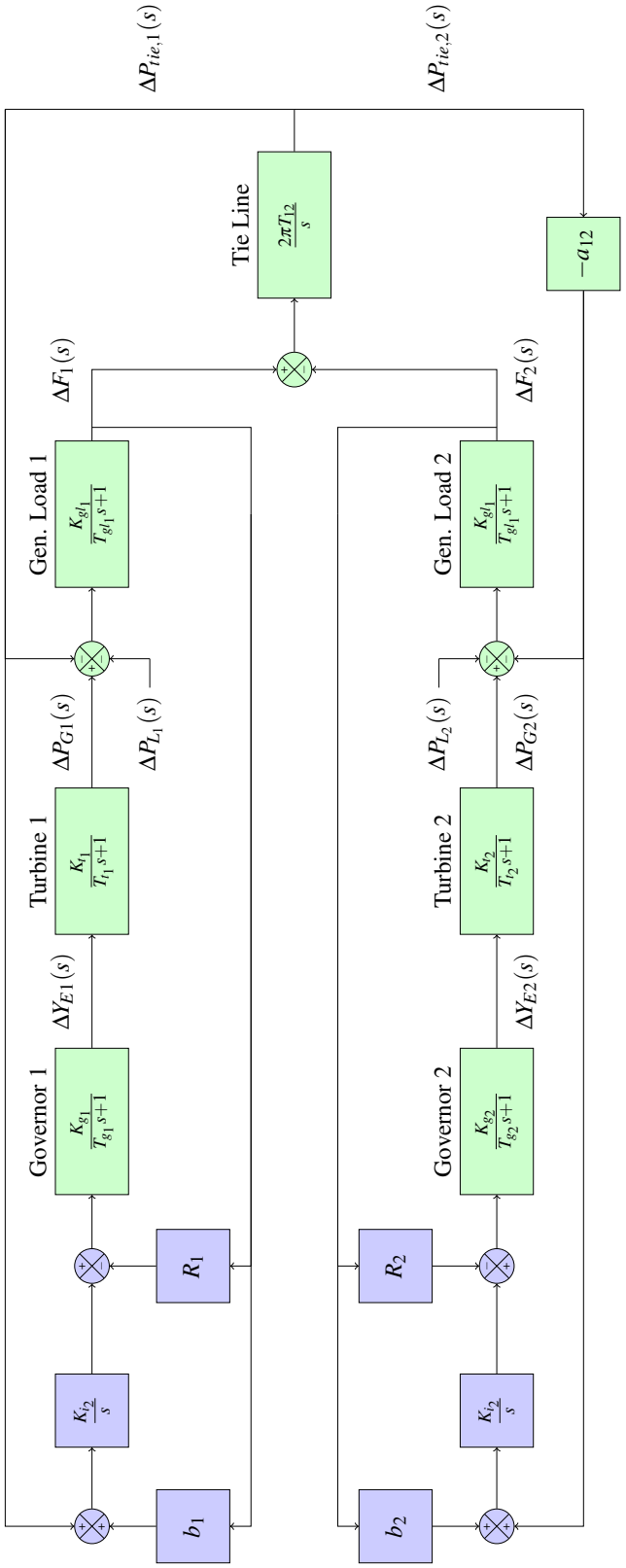

Figure 2.14: A classical feedback control approach to a two area power system [17].

2.2 Reinforcement Learning

According to Sutton and Barto's influential text, RL is a branch of machine learning, based on trial-and-error, that is concerned with sequential decision making [18]. An RL agent exists in an environment. Within the environment it can act, and it can make observations of its state and receive rewards. These two discrete steps, action and observation, are repeated indefinitely with the agent's goal being to make decisions so as to maximise its long-term reward — this scenario is represented in Figure 2.15.

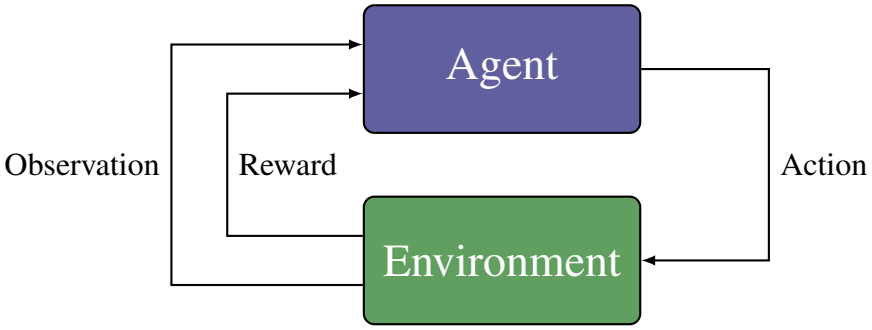

Figure 2.15: Reinforcement learning can be thought of as two discrete steps: action and observation. The agent takes an action and receives a reward. The action causes the agent to change state in the environment.

It is common to represent the RL problem as a set of discrete probabilistic transitions between states, for a set of possible actions that can be selected by the agent. A state transition presents the agent with a reward signal that informs the agent whether an action taken was good or bad. This environmental architecture is referred to as a Markov decision process (MDP). It is the agent's objective to maximise the reward it will receive in the future. An agent can achieve this by learning an optimal policy that maps environment states to actions. Learning such a policy is key idea in RL, and the agent achieves this by experimentation.

2.2.1 Markov Decision Process

Bellman's pioneering work on MPD provided the necessary architecture to develop RL algorithms [19]. His work considered an agent that exists in some environment described by a set of discrete states S. At any discrete point in time the agent can take an action from the set of possible actions A. When the agent takes an action in a given state, the agent receives some reward that is assigned according to a reward function $R : S \times A \times S \rightarrow [R_{min}, R_{max}]$. Fundamental to Bellman's MDPs were the state transition dynamics that were defined by probabilities: if an agent is in a given state, $s \in S$, and takes action, $a \in A$, this will transition the agent to a new state, $s' \in S$, and a yield reward, $r \in R$, with some given probability. This set of probabilities are assigned by a state transition function

$P : S \times A \times S \to [0,1]$. The function P, and it's simpler notation p, is typically written as

$$P(S_{t+1} = s',\ R_{t+1} = r \mid S_t = s,\ A_t = a) = p(s', r \mid s, a). \tag{2.2}$$

Generally, a single reward is bound to a state transition so r is often included as a function argument. The set of parameters outlined above, and Equation 2.2, make up a framework referred to as an MDP.

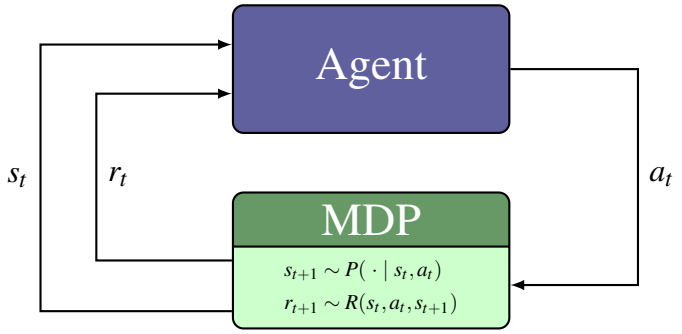

Figure 2.16: Reinforcement learning using an MDP architecture to model state transitions in the environment

2.2.2 Returns, Episodes, and Policy

In addition to developing the MDP framework, Bellman helped develop dynamic programming (DP) [20]. Assuming that the agent has complete knowledge of the state transition probabilities of an environment, DP algorithms can be used to determine analytical solutions for the problem of how an agent should behave to maximise its cumulative reward [20], [21]. This idea was originally thought to be distinct from RL, since DP provides the agent with complete knowledge of its environment, whereas RL agents have no knowledge of environment dynamics and must learn them as well as how to maximise their cumulative reward [18]. Comparisons between DP and RL have been an area of significant interest [22]–[24], but it wasn't until 1989 that Watkins presented the first formal treatment of RL in an MDP framework. Watkin's work showed that DP algorithms could be modified and applied to RL problems [25]. The central ideas in DP algorithms include episodes, returns, and policies [18].

The set of states, actions, and rewards that an agent encounters before arriving at a terminal state is defined as an episode. The set of states and actions (without rewards) is often referred to as the trajectory, τ. It is the agent's goal to take actions such that it maximises the sum of all the rewards at the conclusion of an episode. The cumulative sum of rewards is called the return. Consider an agent taking an action at each discrete time step t, and receiving reward r_t, after each action. If there are N discrete time steps

before the agent reaches a terminal state, Bellman defines the return as:

$$G_t = \sum_{k=0}^{N-1} r_{t+k}. \tag{2.3}$$

Rewards received in the future are often perceived as less valuable than rewards received in the present. To account for this Bellman used a discount factor applied to each reward in the sequence. Letting $\gamma \in [0,1]$ then 2.3 becomes:

$$G_t = \sum_{k=0}^{N-1} \gamma^k r_{t+k}. \tag{2.4}$$

Finally, in order for the agent to take actions it must have a belief of what action it should take, given its current state. This belief is called a policy and is denoted using π [18]. Sutton and Barto define a policy as the mapping of states to actions *i.e.* a rule that determines what actions the agent should take for a given state. A policy can be deterministic, and depend only on the state, $\pi(s)$, or stochastic, $\pi(a|s)$, such that it defines a probability distribution over the actions, for a given state. An optimal policy, denoted π^*, is a policy which will maximise the return an agent receives over an episode.

2.2.3 Value Function and the Bellman Equations

The basic principal of dynamic programming is to assign a value to each state that informs an agent how beneficial a state is to achieving a high cumulative reward. Watkins refers to the creation of systems to assign values to states as the credit assignment problem [25]. Bellman's approach to solving credit assignment was to develop mathematical functions to assign values to states [20]. Bellman's *value function*, $V_\pi(s)$, is defined as the expected sum of the discounted return, G_t, that the agent will receive while following policy π from a particular state s. Mathematically, this is expressed as:

$$V_\pi(s) = \mathbb{E}_\pi\big(G_t \mid s_t = s\big) = \mathbb{E}_\pi\bigg(\sum_{k=0}^{\infty} \gamma^k r_{t+k} \mid s_t = s\bigg). \tag{2.5}$$

The state value function provides a way to check if one policy is better than another policy. It is clear an agent would prefer policy π over some other policy π' if the expected return from using policy π is greater than the expected return from using policy π' for all $s \in S$. Since the state value function is defined by the expected return, Bellman expressed this idea in state value function terms *i.e.* if policy π is preferred to π' then $V_\pi(s) \geq V_{\pi'}(s)$ for all $s \in S$. The optimal policy, π^*, yields the best state value function, $V^*(s)$, referred to as the optimal state value function and defined as:

$$V^*(s) = \max_\pi V_\pi(s),\ \forall\ s \in S. \tag{2.6}$$

Although the state value function provides a way to compare one policy against another policy, it cannot be used to find the optimal policy. Bellman developed a variation of Equation 2.5 called the *state-action value function* that can compare policies and be applied to identify the optimal policy. The state-action value function, $Q_\pi(s,a)$, is defined as the expected sum of the discounted return, G_t, which the agent will receive if it takes action a in state s, and then follows policy π thereafter. Mathematically, this is expressed as:

$$Q_\pi(s,a) = \mathbb{E}_\pi\big(G_t \mid s_t = s,\ a_t = a\big) = \mathbb{E}_\pi\bigg(\sum_{k=0}^{\infty} \gamma^k r_{t+k} \mid s_t = s,\ a_t = a\bigg). \tag{2.7}$$

Similarly to the state value function, the state-action value function's optimal form, $Q^*(s,a)$, can be defined as:

$$Q^*(s,a) = \max_\pi Q_\pi(s,a),\ \forall\ s \in S,\ a \in A. \tag{2.8}$$

To extract the policy π from a state-action value function Q_π the action corresponding to the largest state-action value is chosen for each given state. This is called the greedy policy and is defined, for each $s \in S$, as:

$$\pi(a|s) = \begin{cases} 1 \ \text{ if } \ a = \arg\max_{a'} Q(s,a') \\ 0 \ \text{ if } \ a \neq \arg\max_{a'} Q(s,a') \end{cases} \tag{2.9}$$

Bellman used the value functions presented in Equation 2.5 and 2.7 to formulate recursive expressions which could then be used to solve the DP problem [19]. These are known as the *Bellman equations*. Letting $A(s)$ be the set of actions available in state s, if the agent is operating under the optimal policy π^* then it is true that

$$V^*(s) = \max_{a \in A(s)} Q_{\pi*}(s,a). \tag{2.10}$$

Using the notation shown in Equation 2.2, Equation 2.10 can be rewritten using Equation 2.7

$$V^*(s) = \max_a \sum_{s',r} p(s',r \mid s,a)[r + \gamma V^*(s')]. \tag{2.11}$$

Equation 2.11 is referred to as the Bellman optimality equation for $V^*(s)$. The Bellman optimality equation for $Q^*(s,a)$ is

$$Q^*(s,a) = \sum_{s'} p(s',r \mid s,a)[r + \gamma \max_{a'} Q^*(s',a')]. \tag{2.12}$$

If the transition probabilities and rewards are known to the agent then the Bellman optimality equations can be solved iteratively, which is known as dynamic programming

[19]. Algorithms which assume known transition probabilities and rewards are collectively referred to as *model-based* algorithms. Most RL problems assume state transition probabilities are unknown. The collection of algorithms that provide solutions to these problems are called *model-free* algorithms.

2.2.4 Value Function Based Methods

Model free methods can be applied to any RL problem since they do not require a model of the environment. Algorithms that try to solve the RL problem by estimating the optimal state-action value function, Q^*, and inferring the optimal policy from it are referred to as value function based methods. Figure 2.17 provides an overview of value function based methods. The most common value function based methods are Monte Carlo (MC) and temporal difference approaches.

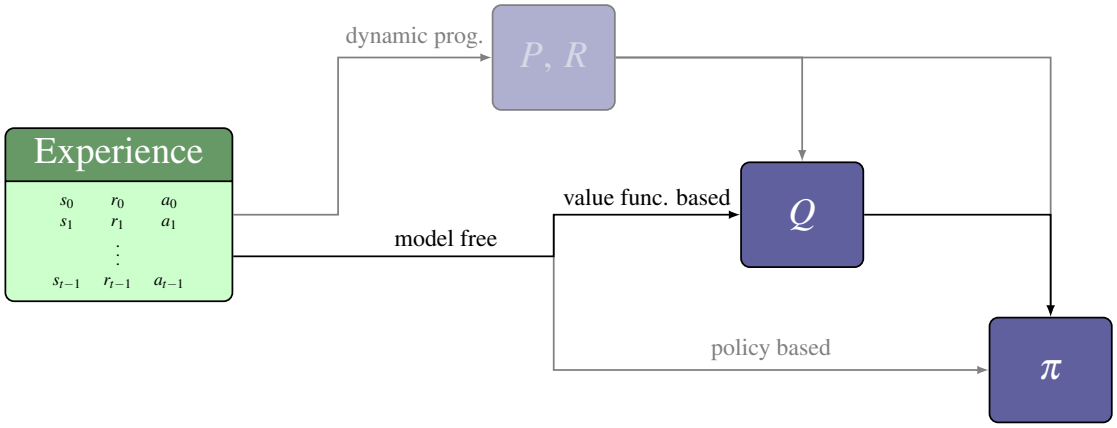

Figure 2.17: Value based approaches try to estimate the state-action value function

Monte Carlo Methods

Sutton and Barto were the first to introduce a version of the MC control algorithm for estimating state-action value functions [18]. The MC control algorithm is based on an iterative approach that takes a sample of episodic sequences consisting of states, actions, and rewards. Sequences are obtained from the agent interacting with the environment, using some policy π. This is called the *policy evaluation* step. Once an episode is completed, the state-action value function, $Q_\pi(s,a)$, is updated for each discrete state-action pair visited. This second step is called the *policy improvement* step.

Sutton and Barto found that during the policy evaluation step, if the agent was allowed to select the action with a greedy policy (Equation 2.9) from the current iteration of the state-action value function, then the MC control would not always converge to the optimal state-action value function. The problem with using the greedy policy is that it can cause the agent to over commit to locally promising but globally poor solutions. This is

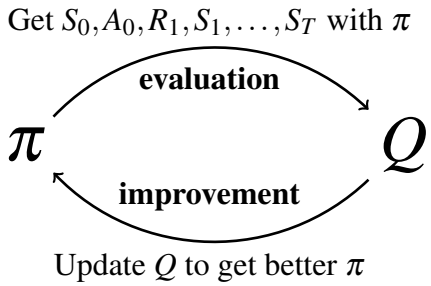

Figure 2.18: The Monte Carlo algorithm takes a policy evaluation step and policy improvement step, for each episodic iteration.

due to the agent not sufficiently exploring the state-action value space, and is especially problematic during the early stages of the algorithm where the agent has little knowledge about the environment. This problem is well known in RL research and is referred to as the exploration-exploitation dilemma.

The most common method of overcoming this problem for MC is to use an ε-greedy policy instead of the greedy policy. The idea behind the ε-greedy policy is to select the greedy action most of the time, and select non-greedy actions the other times. This is achieved by setting a parameter, $\varepsilon \in [0,1]$, which allows the agent to select non-greedy actions with a non-zero probability. The ε-greedy policy is defined as:

$$\pi(a|s) = \begin{cases} 1-\varepsilon & \text{if } a = \arg\max_{a'} Q(s,a') \\ \frac{\varepsilon}{|A|-1} & \text{if } a \neq \arg\max_{a'} Q(s,a'). \end{cases} \tag{2.13}$$

Originally, for a given state-action pair, (s,a), the policy improvement step updated the state-action value function, $Q(s,a)$, using an average approach. The average value used for the update was derived from the returns of all visits, past and present, to a given state-action value pair. This approach worked; however, learning in the later stages of the algorithm was reduced because the averaging algorithm placed a uniform importance on all stages of learning. To overcome this problem, Sutton and Barto employed a coarser update rule using the difference between observed returns and the current state-action value function. Multiplying the update by some value $\alpha \in [0,1]$ ensures update steps are not too large. If update steps are too large, this may prevent the algorithm convergence to Q^*. For some point in time t, if S_t is the state, A_t is the action, and G_t is the observed return for completion of the current episode, then update rule can be written as:

$$Q(S_t,A_t) \leftarrow Q(S_t,A_t) + \alpha[G_t - Q(S_t,A_t)] \tag{2.14}$$

The MC implementation using the ε-greedy policy 2.13 for the evaluation step, and the constant α update rule in 2.14 is referred to as the onstant α MC control algorithm. The algorithm is shown in listing 1. Note that the algorithm returns as approximation of

the optimal state-action value function, and the optimal policy can be extracted using the greedy policy in 2.9.

Algorithm 1 Constant α Monte Carlo Control

Input: `num_episodes`, α, ε_i
Output: Q ($\approx Q^*$ if `num_episodes` is large enough)
Initialise Q such that $Q(s,a) = 0$ for all $s \in A$ and $a \in A$
for $i \leftarrow 1$: `num_episodes` **do**
    $\varepsilon \leftarrow \varepsilon_i$
    $\pi \leftarrow \varepsilon$ – greedy
    Generate and episode $S_0, A_0, R_1, \ldots, S_T$ using π
    for $t \leftarrow 0 : (T-1)$ **do**
        if (S_t, A_t) is a first visit **then**
            $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha(G_t - Q(S_t, A_t))$
        end if
    end for
end for
return Q

Temporal Difference Methods

Monte Carlo methods collect experience from an episode, and update the policy once the episode is complete. This approach works for environments that have short episodes, but becomes computationally intractable for environments where an episode takes a long time to reach a terminal state, or for continuing tasks that never reach a terminal state. Temporal difference (TD) methods address this problem.

TD methods take an almost identical approach to MC methods, except they perform state-action value function updates after every time step during an episode instead of waiting until episode completion. This is achieved by estimating the return for a given time step, since it would be unknown during an episode. More concretely, for a given state-action pair, (S_t, A_t), which transitions the environment to state S_{t+1}, the return is estimated using the transition reward, R_{t+1}, and the state-action value function estimate for the subsequent state, assuming a greedy policy:

$$G_t \approx R_{t+1} + \max_a Q(S_{t+1}, a) \tag{2.15}$$

Along with his highly influential integration of RL with MDPs and DP, Watkins developed one of the most widely used TD algorithms called Q-learning [25]. Watkins used 2.15 to modify the update rule 2.15 as follows:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)] \tag{2.16}$$

The Q-learning algorithm is summarised in listing 2.

Algorithm 2 Q-learning

Input: `num_episodes`, α, ε_i
Output: value function Q ($\approx Q^*$ if `num_episodes` is large enough)
Initialise Q such that $Q(s,a) = 0$ for all $s \in S$ and $a \in A$
for $i \leftarrow 1$: `num_episodes` **do**
    $\varepsilon \leftarrow \varepsilon_i$
    Observe initial state S_0
    $t \leftarrow 0$
    repeat
        Select an action A_t using policy derived from Q (e.g. ε-greedy)
        Carry out action A_t
        Observe reward R_{t+1} and new state S_{t+1}
        $Q(S_t,A_t) \leftarrow Q(S_t,A_t) + \alpha[R_{t+1} + \gamma \max_a Q(S_{t+1},a) - Q(S_t,A_t)]$
        $t \leftarrow t+1$
    until S_t is terminal
end for
return Q

2.2.5 Policy Search Methods

Policy search methods are another class of RL algorithms that do not use value functions to determine state to action policy mappings. Figure 2.19 provides an overview of policy search methods compared to dynamic programming and value function based methods.

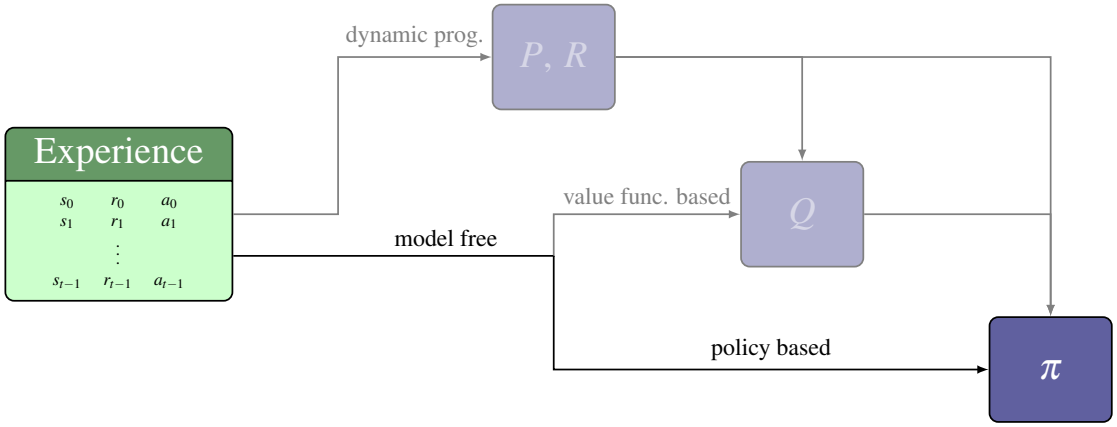

Figure 2.19: Policy search approaches try to estimate the policy directly without using state-action values.

Instead of value functions, policy search methods use parameterised policies, $\pi(a|s;\theta)$. Parameters are represented as a vector, $\theta \in \Theta$, and changing individual elements of vector θ changes how the policy maps states to actions. The novelty of this approach is that the policy is expressed as a parameterised functional form, where the function can be anything from a linear model to a neural network, although not all functions perform equally.

With policy search, the agent searches directly for the optimal policy in vector space, Θ. To express this idea more formally, let $J(\theta)$ denote the expected value of the discounted

return of trajectory τ, for policy $\pi(a|s;\theta)$. Mathematically, this is written as:

$$J(\theta) = \mathbb{E}_{\tau\sim\pi(a|s;\theta)}\left(\sum_{t=1}^{\infty}\gamma^{t-1}r_t\right). \tag{2.17}$$

Finding the optimal policy, for policy search, can now be expressed as a maximisation problem of $J(\theta)$, *that is*:

$$\pi^* = \pi\big(a|s;\arg\max_{\theta} J(\theta)\big) \tag{2.18}$$

Expressing the policy search problem this way allows for the application of many different optimisation algorithms. One simple approach is to use a hill climbing algorithm. Suppose the agent possess some policy, π_θ, which it can roll-out in the environment and get some return, G. It can then slightly modify θ to get some marginally different policy, say $\pi_{\theta'}$. The agent could then roll-out the new policy in the environment and receive a return G'. If $G' > G$, then hill climbing would discard policy π_θ in favour of $\pi_{\theta'}$; and if $G' < G$, then hill climbing retains policy π_θ. Continued iterations of the above would see the agent's policy converge to an optimal policy, although this is not guaranteed to be globally optimal.

Another algorithm that is used to solve Equation 2.18 is gradient ascent. In gradient ascent, the parameter update direction is given by the gradient $\nabla_\theta J(\theta)$ as it points in the direction of steepest ascent of the expected return. Provided $\nabla_\theta J(\theta)$ exists, the parameter update rule is expressed as:

$$\theta \leftarrow \theta + \alpha\nabla_\theta J(\theta), \tag{2.19}$$

where α is a user-specified learning rate.

Policy search generally has better convergence properties and is also able to learn stochastic policies which are not possible with value based approaches [26]. The biggest drawback of policy search algorithms is their policy evaluation step, which can suffer from a large variance resulting in policy search agents taking a long time to learn optimal policies.

2.3 Deep Neural Networks

Deep neural networks (DNNs) are responsible for some of the most recent state-of-the-art technological breakthroughs in fields such as audio to text speech recognition systems [27], image classification systems [28]–[31], text-to-text machine translation, and robotics [32]–[35].

Deep neural networks are able to adapt to the different needs of diverse research fields due to their unique computational architecture.

2.3.1 Feedforward Networks

The most common architecture used in a DNN is the *feedforward* neural network (FNN), often referred to as a *multilayer perceptron* model. First developed in 1971, a fully connected FNN consists of an input layer, one or more hidden layers, and an output layer, as shown in Figure 2.20.

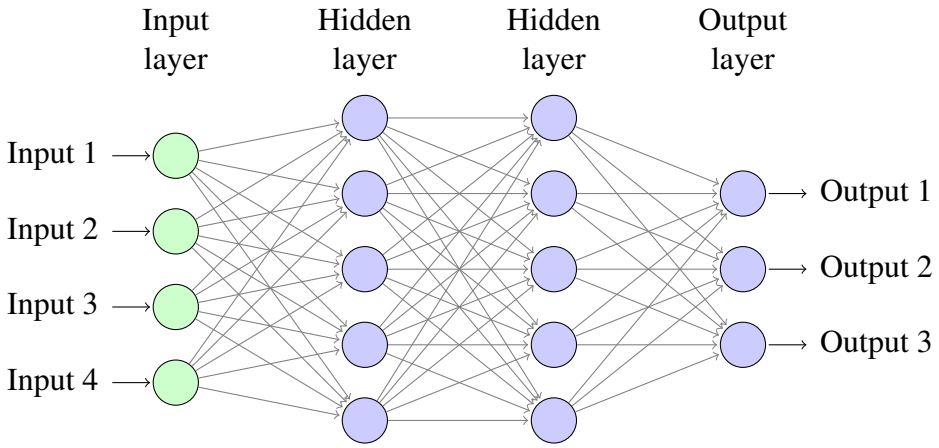

Figure 2.20: An example of a feedforward network consisting of an input layer, two hidden layers, and an output layer

When an input signal is introduced to the input layer, it propagates through the network, in a forward direction, on a layer-by-layer basis. The emergent signal at the output layer, is called the network response [36]. Hidden and output layers are made up of multiple nodes, called perceptrons. Each perceptron receives a vector of inputs from the previous layer. Weights are applied to each vector element. A summation of the weighted vector elements is passed through an activation function, that allows the node to signal when it recognises the vector input. This is discussed further in §2.3.2.

The weighting values described above allow FNN architectures to connect many perceptrons to form a computational graph, or network. Modifying the network weights in a way that allows the FNN to achieve the desired behaviour is referred to as *training the network*. A trained network is able to form highly non-linear models used for estimation or classifying of complex phenomena that is difficult to model using classical approaches, or is computationally intractable.

2.3.2 Perceptron Model

Rosenblatt is credited with developing the perceptron model that is a fundamental building block for neural network architectures. Motivated by Hebbian theory of synaptic plasticity (*i.e.* the adaptation of brain neurons during the learning process), Rosenblatt developed a model to emulate the "perceptual processes of a biological brain" [37]. Rosenblatt's perceptron model consisted of a single node used for binary classification of patterns that

are linearly separable [38]. Letting input vector elements be x_i, weight terms be w_i, and the bias term be b, the summation operation can be expressed as:

$$\sum_i x_i w_i + b. \tag{2.20}$$

The summation is then passed through an activation function, f, to produce the neuron output. Using equation 2.20 and letting the neuron output be y, the neuron model can be expressed as:

$$y = f\left(\sum_i x_i w_i + b\right). \tag{2.21}$$

Figure 2.21 shows the computational model of a neuron, expressed in Equation 2.21.

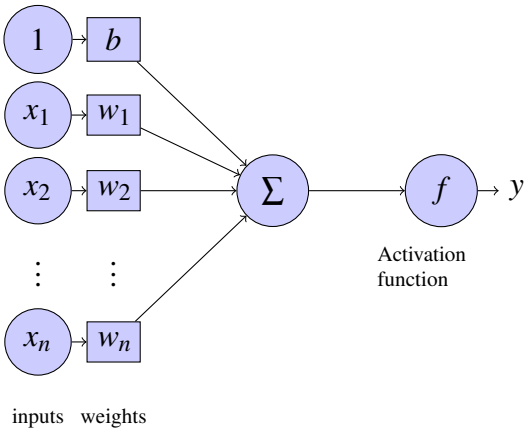

Figure 2.21: Rosenblatt's perceptron model passes the summation of weighted inputs to an activation function

2.3.3 Activation Functions

The activation function is a key component of Rosenblatt's perceptron — it determines if the perceptron will activate or not. Rosenblatt's model used a Heaviside step function, which was effective under his perceptron learning algorithm for some classification tasks. In 1969, Minsky and Papert published a book called Perceptrons that argued Rosenblatt's perceptron had significant limitations, such as the inability to solve exclusive-or (XOR) classification problems [39]. This limitation was overcome by increasing the size of neural networks, which subsequently required the development of new algorithms to train the networks efficiently. Gradient descent using backpropagation [40] is the most common algorithm for training neural networks. These algorithms are discussed further in §2.3.4; however, one important requirement is that activation functions are differentiable, with non-zero gradients. The Heaviside function is non-differentiable at $x = 0$, and has a zero derivative elsewhere and is therefore not suitable. According to Haykin, a number of different activation functions can be adopted to replace the Heaviside function [36]. Two of

the most common are:

1. Hyperbolic tangent: $f : \mathbb{R} \to (-1, 1)$, shown in Figure 2.22, where

$$f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{2.22}$$

2. Logistic sigmoid: $f : \mathbb{R} \to (0, 1)$, shown in Figure 2.23, where

$$f(x) = \frac{1}{1 + e^{-x}} \tag{2.23}$$

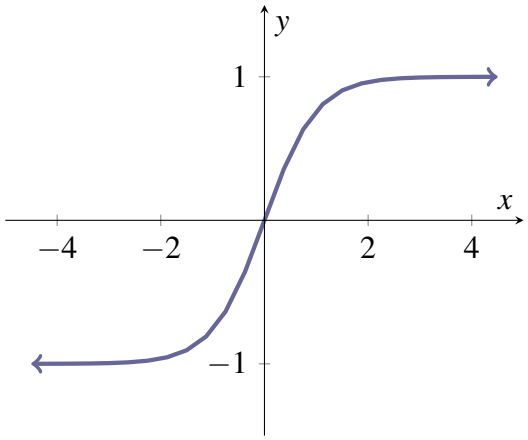

Figure 2.22: Hyperbolic tangent activation function

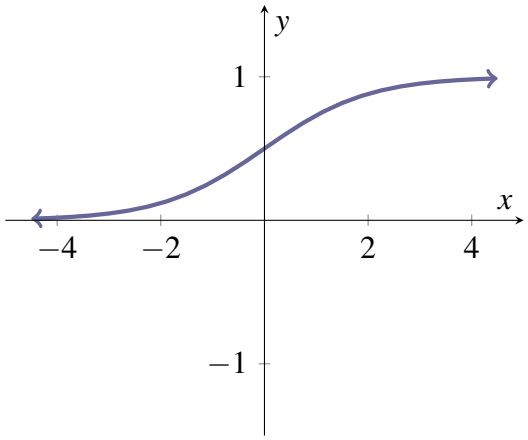

Figure 2.23: Sigmoid activation function

For shallow neural networks, the hyperbolic tangent and sigmoid activation functions are effective. As the network is deepened with additional hidden layers hyperbolic tangent and sigmoid activation functions cause the network to learn sub-optimally. This phenomena was first discovered by Hochreiter in 1991 and is referred to as the vanishing gradient problem [41]. It describes a situation where the gradients for hyperbolic tangent and sigmoid activation functions become close to zero toward the function tails. As previously mentioned, training algorithms such as gradient descent require non-zero gradients. As inputs to sigmoidal or hyperbolic tangent functions get very large (or very small) the gradient tends to zero. This impedes the agent's ability to learn. Two commonly used activation functions that are used to overcome the vanishing gradient problem are:

1. Rectified Linear Unit (ReLU): $f : \mathbb{R} \to [0, \infty)$, shown in Figure 2.22, where

$$f(x) = \max(0, x) = \begin{cases} x & \text{if } x \geq 0 \\ 0 & \text{if } x < 0 \end{cases} \tag{2.24}$$

2. Leaky ReLU (LReLU): $f : \mathbb{R} \to \mathbb{R}$, shown in Figure 2.23, where for $\alpha < 1$

$$f(x) = \max(x, \alpha x) = \begin{cases} x & \text{if } x \geq 0 \\ \alpha x & \text{if } x < 0 \end{cases} \tag{2.25}$$

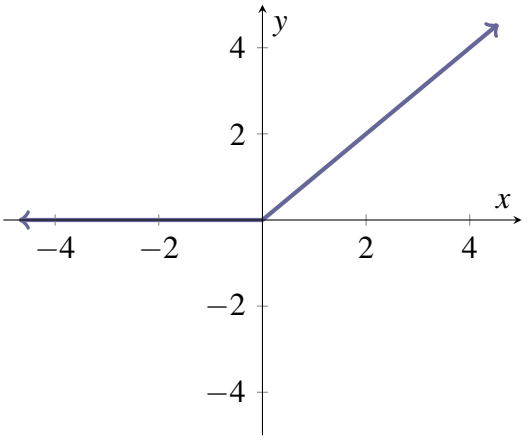

Figure 2.24: ReLU activation function

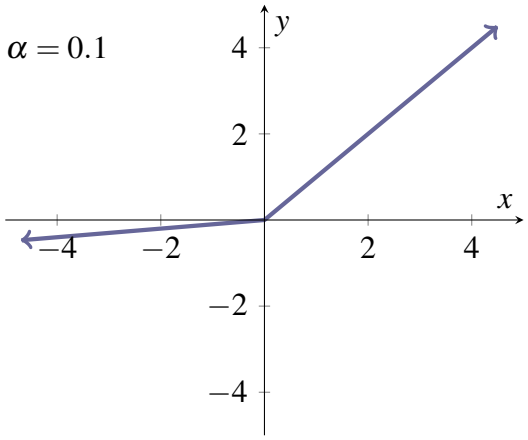

Figure 2.25: LReLU activation function

2.3.4 Training the Network

As outlined in §2.3.1, modifying the weights that are multiplicatively applied to perceptron inputs changes the perceptron's behaviour. Moreover, changing the weights of perceptrons in a neural network changes the behaviour of the network. Changing the weights in a systematic way to shift network behaviour towards the desired result is known as training the network.

The most common way of training the network is by using the Gradient Descent optimisation algorithm, which minimises a cost function. A neural network cost function quantifies the error between the network's prediction and an actual, or expected, result for a set of training data. Often the Mean Squared Error is used as a cost function, although cost function specification is dependent on the neural network application. Letting θ represent the set of network weights, a neural network cost function is typically denoted as, $L(\theta)$.

The Gradient Descent algorithm calculates the gradient of the cost function with respect to network weights. This is denoted by $\nabla_\theta L(\theta)$. The gradient represents the direction of the steepest slope on the cost function manifold. For the $k+1$th iteration, the algorithm takes a small step, $\alpha \nabla_\theta L(\theta)$, in the opposite direction of the gradient, where α is referred to as the learning rate. The update equation for the network weights at $k+1$th step is

given by:

$$\theta_{k+1} = \theta_k - \alpha \nabla_\theta L(\theta) \tag{2.26}$$

The gradient calculation and update steps are repeated iteratively until the algorithm converges on a local or global minima.

Efficient calculation of the neural network cost function gradients is achieved using the an important algorithm called backpropagation. Backpropagation uses computational graph structures and the chain rule to recursively calculate gradients, with respect to network weights, throughout the entire neural network. The backpropagation algorithm was the first algorithm to achieve this result efficiently, and remains the workhorse of deep neural network learning.

2.4 Deep Reinforcement Learning

Deep reinforcement learning (DRL) refers to the use of deep neural networks in reinforcement learning frameworks. In this context, neural networks have been used successfully with the Q-learning algorithm as state-action value function approximators. Additionally, they have also seen success when used as function approximators for the policy directly. This section discusses two of the most prominent DRL algorithms in use today: Deep Q-learning, and the Deep Deterministic Policy Gradient.

2.4.1 Deep Q-Learning

Reinforcement learning algorithms outlined in §2.2 are capable of controlling environments with low dimensional state space representations, but perform poorly in environments with high dimensional state space representations. In 2015 Mnih *et alias* published an influential paper that presented a novel agent: a Deep Q-network (DQN) [32]. The agent was able to take advantage of recent developments in the field of deep neural networks, discussed in §2.3, and combine them with the Q-learning algorithm, discussed in §2.2. By adapting the Q-learning algorithm, Mnih *et al.* trained a neural network, $Q(s,a|\theta)$, to act as a function approximator for the state-action value function $Q(s,a)$. This approach was able to train agents that achieved human levels of performance in video games.

Historically, the applications of neural networks to reinforcement learning have performed poorly because they were unstable and divergent. One of the reasons for this is because of temporally correlated sequences of states, actions, and rewards arising during an episode causing over fitting in the network during training [42]. Mnih *et al.* overcame the temporal correlations problem by creating a buffer, D, to store agent experience, (s_t, a_t, r_t, s_{t+1}). At each time step, experience is recalled from the buffer via uniform random sampling. The samples are then used to train the network. The technique is referred

to as *experience replay*.

The other main innovation from DQN was the use of a second neural network called the *target network*, represented by $Q(s,a|\theta^-)$. The target network parameters, θ^-, are a copy of the original network parameters θ, that are updated periodically. Letting the notation $(s,a,r,s') \sim U(D)$ denote uniform random sampling from the buffer, the DQN loss function can be expressed as the mean-squared error of the Q-learning (2.16) temporal difference update:

$$L_i(\theta_i) = \mathbb{E}_{(s,a,r,s')\sim U(D)}\left[\left(r + \gamma \max_{a'} Q(s',a'|\theta_i^-) - Q(s,a|\theta_i)\right)^2\right] \tag{2.27}$$

Applying gradient descent using backpropagation to Equation 2.27 adjusts network weights, as described in §2.3.4. If the target network is not used then a correlation arises between the state-action values $Q(s,a|\theta)$ and the target values, $r + \gamma \max_{a'} Q(s',a'|\theta)$, causing unstable learning, similar to the temporal correlation problem described above.

The DQN algorithm is summarised in listing 3.

Algorithm 3 DQN Algorithm

1: Initialise replay memory D to capacity N
2: Initialise action-value function Q with random weights θ
3: Initialise target action-value function Q with weights $\theta^- = \theta$
4: **for** $episode \leftarrow 1 : M$ **do**
5: Receive initial observation state s_1
6: **for** $t \leftarrow 1 : T$ **do**
7: With probability ε select a random action a_t otherwise select $a_t = \arg\max_a Q(s_t, a; \theta)$
8: Execute action a_t and observe reward r_t and state s_{t+1}
9: Store transition (s_t, a_t, r_t, s_{t+1}) in D
10: Sample random minibatch of transitions (s_j, a_j, r_j, s_{j+1}) from D
11: Set $y_j = \begin{cases} r_j & \text{if episode terminates at step } j+1 \\ r_j + \gamma \max_{a'} Q(s_{j+1}, a'|\theta^-) & \text{otherwise} \end{cases}$
12: Perform a gradient descent step on $\left(y_j - Q(s_j, a_j|\theta)\right)^2$ with respect to network parameters θ
13: Every C steps reset $\theta^- = \theta$
14: **end for**
15: **end for**

The DQN algorithm is able to train neural networks to solve problems with high-dimensional state spaces; however, it is only suitable for discrete, low-dimensional actions spaces. This is because the algorithm needs to calculate the maximum over actions during the update step, as shown in line 11 of Listing 3. The implication being that each update step would be computationally expensive for action spaces with a large number of discrete actions, and computationally intractable for continuous action spaces.

2.4.2 Deep Deterministic Policy Gradient

In late 2015 Lillicrap *et al.* adapted key ideas from DQN and the deterministic policy gradient theorem from [43] and developed the first algorithm that could train a neural network successfully on problems with continuous action domains [33]. The algorithm is called Deep Deterministic Policy Gradient (DDPG), and uses a total of four neural networks. The first network is called the actor, $\pi(s|\theta^{\pi})$, where θ^{π} denotes the network parameters. The actor part of the DDPG agent is classified as a policy search method, discussed in §2.2.5. The second network is called the critic, $Q(s,a|\theta^{Q})$, where θ^{Q} denotes the network parameters. The critic part of the DDPG agent is classified as a value function method, discussed in §2.2.4.

DDPG makes use of DQN's target network idea, implementing a further two neural networks — one for each of the actor and critic networks. The network parameters for the actor and critic target networks are denoted $\theta^{\pi'}$ and $\theta^{Q'}$, respectively. DDPG also makes use DQN's experience replay buffer to store experience which is randomly sampled from during training.

The loss function for the critic network is similar to the DQN loss function (2.27) except that actions are selected by the actor network. Using the standard Q-learning update and the mean square error, the critic loss function is expressed as:

$$L_i(\theta_i^{Q}) = \mathbb{E}_{(s,a,r,s')\sim U(D)}\left[\left(r+\gamma Q\big(s',\pi\big(s'|\theta_i^{\pi'}\big)|\theta_i^{Q'}\big)-Q\big(s,a|\theta_i^{Q}\big)\right)^2\right] \tag{2.28}$$

The actor network is updated using the deterministic policy gradient theorem [43]. The gradient update is given by:

$$\nabla_{\theta^{\pi}} J(\theta^{\pi}) = \mathbb{E}\big[\nabla_a Q(s,a|\theta^{Q})|_{s,\ a=\pi(s|\theta^{\pi})} \nabla_{\theta^{\pi}} \pi(s|\theta^{\pi}) \mid s\big] \tag{2.29}$$

Equations 2.28 and 2.29 are used with gradient descent and the backpropagation algorithms to update actor and critic network weights during training. The full DDPG algorithm can be seen in listing 4.

Using DDPG, Lillicrap *et al.* were able to solve challenging problems across a variety of domains with continuous action spaces. Moreover, DDPG was shown to be more efficient that DQN for the video game domain, needing fewer steps of experience than DQN to learn solutions.

Algorithm 4 DDPG Algorithm

Randomly initialise critic $Q(s,a|\theta^Q)$ and actor $\mu(s|\theta^Q)$ with weights θ^Q and θ^μ
Initialise target networks $Q\prime$ and $\mu\prime$ with weights $\theta^{Q'} \leftarrow \theta^Q$, $\theta^{\mu'} \leftarrow \theta^\mu$
Initialise replay buffer R
for $episode \leftarrow 1 : M$ **do**
    Initialise a random process $\mathcal{N}$ for action exploration
    Receive initial observation state s_1
    for $t \leftarrow 0 : (T-1)$ **do**
        Select action $a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t$ with noise exploration noise
        Execute action a_t and observe reward r_t and observe new state s_{t+1}
        Store transition (s_t,a_t,r_t,s_{t+1}) in R
        Sample a random minibatch of N transitions (s_i,a_i,r_i,s_{i+1}) from R
        Set $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{Q'}))$
        Update critic by minimising the loss: $L = \frac{1}{N}\sum_i (y_i - Q(s_i,a_i|\theta^Q))^2$
        Update the actor policy using the sampled policy gradient:

$$\nabla_{\theta^\mu} J \approx \frac{1}{N}\sum_i \nabla_a Q(s,a|\theta^Q)|_{s=s_i,a=\mu(s_i)} \nabla_{\theta^\mu}\mu(s|\theta^\mu)|_{s=s_i}$$

        Update the target networks:

$$\theta^{Q'} = \tau\theta^Q + (1-\tau)\theta^{Q'}$$
$$\theta^{\mu'} = \tau\theta^\mu + (1-\tau)\theta^{\mu'}$$

    end for
end for

Chapter 3

Literature Review

To appreciate historical developments in the field of load frequency control, this chapter first considers industry standard solutions, and the motivations for their continued use. Following this, alternative control approaches are considered including fuzzy logic control, genetic algorithms, and artificial neural networks. The chapter concludes with an analysis deep reinforcement learning applied to the load frequency control problem.

3.1 Classical Controllers

The first attempts at power system frequency control was using a flywheel governor attached to the synchronous machines [44]. Originally, as discussed in §2.1.1, turbine governors were mechanically coupled to generator shafts providing a built-in proportional feedback control loop. As the frequency deviated from the generator's set point, the governor would open or close a steam valve to increase or decrease the rotational speed accordingly. This type of control is often referred to as the primary control loop [16], as discussed in §2.1.2. Early synchronous machine operators found that primary loop governor control would require constant tuning to ensure that the power system was operating at the scheduled frequency [45]. Analysis undertaken using first-order linear models for the governor, turbine, and generation load control confirmed that governor primary loop control was able to arrest frequency deviations; however, was unable to return the system to scheduled operating frequencies [46]. Subsequently, a secondary feedback control loop was introduced which used the integral of frequency deviations, as discussed in §2.1.4. The PI control scheme constitutes the classical approach to solving the load frequency control problem [44].

Throughout the 1930s and 1940s, North American utility providers applied PI architectures to control the frequency of interconnected power systems comprised of two or more areas [45]. As detailed in §2.1.4, the aim was to regulate the frequency in each area in addition to preserving the power flow over the tie-lines interconnecting power system

areas. In 1953, Concordia and Kirchmayer were first to present an empirical analysis on the dynamic performance of a PI controller tasked with the frequency control of a two area power system. They concluded that optimum control gains for the system existed; however, they could not determine these analytically [47].

Cohn [48] and Aggarwal *et al.* [49], [50] undertook pioneering work to develop classical control approaches to work with power systems comprised of two or more control areas. Their work focused on controllers which were able to maintain the system frequency at scheduled values in multiple power areas, as well as maintain scheduled power flow over transmission infrastructure (tie lines) connecting the areas. Cohn's paper used a first-order linear system to model the tie line dynamics. The development of a tie line model allowed Cohn to use PI control architectures for each power system area, albeit with modified input signals. This work led to the development of the feedback signal called area control error (ACE). Cohn used ACE to ensure power systems were restored to the scheduled frequency, given a frequency deviation, and that unscheduled tie line power flows were minimised between neighbouring control areas [51]. Elgerd and Fosha were the first to deliver an optimal control concept for frequency control design of interconnected power systems [52].

Classical power system frequency controllers remain a popular choice for industry given they are robust and simple and ...

They can be easily designed using Bode and Nyquist diagrams to obtain desired gain and phase margins. Root locus plots can also be used [53]. While these approaches are simple, well known, and easy for practical implementation, investigations using these approaches have resulted in control schemes that exhibit poor dynamic performance in the presence of parameter variations and nonlinearities [17], [52], [54].

Frequency controller analysis and design assumes plant models have linear dynamics; however, studies have shown that modern power systems display complex non-linear dynamics [55]–[58]. Modern power systems are large-scale and comprised of multiple power generation sources such as thermal, hydro, and photovoltaic power. Commonly researched generator non-linearity include the effects of governor dead band (GDB) [55] and generator ramp constraints (GRC) [56], [57]. Moreover, modern power systems use high voltage direct current (HVDC) lines to export power over long distances, and they also feature energy storage systems such as pumped hydro or batteries [12], [13], [16], [17]. Both of these features display highly non-linear characteristics.

Linear ODE power system models capture underlying plant characteristics; however, these models are only valid within certain operating ranges. Non-linear plant characteristics mean that different linear ODE models are required as plant operating conditions change. Governor dead band is observed as a change in generator angular velocity for which there is no change in the governor valve position. GDB is generally attributed to backlash in the governor mechanism, and degrades LFC performance. GRC is a physical

limitation of the turbine that imposes upper and lower boundaries on the rate of change in generating power from the turbine [58]. In recent years, frequency control methods using fuzzy logic, genetic algorithms (GAs), and artificial neural networks (ANNs), have attempted to address the problems that arise due to non-linearity.

3.2 Fuzzy Logic Control

Fuzzy logic control schemes are developed directly from power system domain experts or operators who control plant manually. Researchers have shown that a fuzzy gain scheduling PI controller can perform as well as a fixed gain controller, for frequency control of two and multi-area power systems. Additionally, it has been found that fuzzy controllers are simpler to implement [59], [60]. Yesil *et al.* [61] proposed a self-tuning fuzzy PID controller for a two-area power system and noted improvements in controller transient performance when compared to a fuzzy gain scheduling PI controller.

3.3 Genetic Algorithms

Genetic algorithms are stochastic global search algorithms based on natural selection. In the context of power system control, GAs operate on a population of individuals. An individual is a set of control system parameters that are initially drawn at random and without knowledge of the task domain. Successive generations of individuals are developed using genetic operations such as recombination or mutation. The chance of an individual being selected for use in a genetic operation is based on an objective measure of fitness — strong individuals are retained and weak individuals are discarded [62].

Chang *et al.* [63] investigated using GA to determine fuzzy PI controller gains, which resulted in a control scheme which performed favourably when compared to a fixed-gain controller. Rekpreedapong *et al.* [64] took this further by optimising PI controller gains with GA while using linear matrix inequalities (LMI) constraints from a higher order controller. This research, performed on a three-area control system, was motivated by the belief higher order controllers are not practical for industry. Rekpreedapong *et al.* concluded that the GA tuned PI controller, under LMI constraints, performed almost as well as a higher order control system. Research undertaken by Ghosal [65] concluded that PID control with gains optimised by GA provided better transient performance than PI control with gains optimised in the same way.

3.4 Neural Networks

Neural networks are systems that take input signals and by using many simple processing elements, produce output signals, as discussed in §2.3.

Beaufays *et al.* [66] demonstrated it was possible to used a neural network for frequency control in one and two-area power systems. A neural network was used instead of an integral controller in the classical structure and received a state variable input vector containing frequency deviation and tie-line power measurements instead of a single value ACE signal seen with classical controllers. The network was trained using gradient descent and the back propagation algorithms, resulting in better transient performance when compared with a classical PI controller. Using these results, Demiroren et al. [67] went further by including non-linearity in the plant models. Specifically, governor deadband, reheater effects, and generating rate constraints are included and it was shown that the results obtained using the ANN controller outperformed the results of a standard PI classical control model for a two-area power system. Additional work confirmed these results for a larger four-area power system with thermal and hydro generation sources [68].

3.5 Deep Reinforcement Learning

A literature survey on deep reinforcement learning for power system applications, published in March 2020, highlighted that load frequency control is a critical aspect of operational control that is becoming more complex and challenging as renewable energy sources become more prevalent. Moreover, the survey only cited one study which focused on using DRL in the continuous action space for load frequency control [69].

The study, published in 2019 by Yan and Xu, used a DDPG trained neural network to control the frequency of a single simulated power area consisting of a non-reheat steam turbine, wind turbine, and stochastic load demand. Yan and Xu concluded that the DDPG trained neural network was able to provide control action that resulted in reduced frequency deviation when compared to an optimally tuned PI controller [70].

Chapter 4

Approach

This chapter presents the experimental approach which is broken up into two phases, undertaken sequentially. The first phase focuses on development, implementation, and verification of the software implementations for the power system environment model, controllers, and neural network training algorithms. The second phase uses the developed models to experimentally assess the performance of a neural network for load frequency control using different neural network architectures, hyperparameters, expert training examples, and stochastic load demand profiles.

4.1 Development, Implementation, and Testing

4.1.1 Environment and PI Controller

Training a neural network to act as a load frequency controller using DDPG requires a simulated model of a two area power system. This will be referred to as the environment model. The environment model developed by this research will consist of two power areas connected via a tie-line, each with a governor, turbine, and generator. The model will take 2 control action inputs — one for each power system area. The model will be designed to output 7 values including governor states, turbine states, and power system frequencies for each area, in addition to the power transfer of the tie-line.

The mathematical representation of the environment model will be developed in the temporal domain. This design choice is motivated by the need to stop and start the simulation periodically, allowing the neural network to take simulation states as inputs, and issue control action outputs. Upon receiving a control action the simulation will iterate forward by some discrete time interval, using previous stopping states as initial conditions. This could not be achieved in the frequency domain as initial conditions are assumed to be zero. The temporal domain model will be developed by taking inverse Laplace transforms of frequency domain models from simulation experiments discussed in the literature review.

A PI controller model will be developed to provide a performance benchmark compar-

ison for the neural network. The model will take 3 inputs from the environment model, namely the frequency from each power system area, and the tie-line power transfer. Whilst the PI controller could be developed in the frequency domain, for convenience it will be developed in the temporal domain to interact with the temporal environment model of the two area power system. The PI controller model will be mathematically specified using the same methodology described for the two area power system environment model.

Model parameter values for the two area power system model will be selected based on a review of the most common model parameters observed in the literature review. Tuned model parameters for the PI controller will selected to ensure near optimal control performance.

4.1.2 Neural Network and DDPG Algorithm

Neural network actor-critic models will be designed to have an input layer, 2 hidden layers, and an output layer. This design choice was motivated by Lillicrap *et alias* [33]. The input layer will be designed to receive 7 model inputs, consisting of governor and turbine states for each power system area, frequencies for each power system area, and the power transfer on the tie-line. The output layer will issue 2 control actions to be received by the environment model.

The neural network model will be developed to allow for modification to the number of nodes and activation functions in hidden layers. This design choice has been made to facilitate investigation of neural network control performance with respect to model architecture changes based on the work by Henderson *et alias* [71].

The DDPG algorithm will be developed based on Algorithm 4, shown in section 2.4.2. Algorithm hyperparameters, including γ, τ, $\alpha_{\texttt{actor}}$, $\alpha_{\texttt{critic}}$ will be specified according to original experiments undertaken by Lillicrap *et alias* [33]. Parameters such as batch size, replay buffer size, number of training episodes will be selected based on work done by Yan [70]. Values will be slightly modified according to observations from minor preliminary experimentation with the power system environment model, neural network model, and training algorithm implementations. Justifications will be provided where parameters are modified.

The DDPG algorithm will be developed with Ornstein-Uhlenbeck (OU) noise superimposed on control actions, in order to explore the state-action space as per Lillicrap *et alias* [33]. The OU noise process will be designed to allow easy modification of noise parameters μ, θ, and σ. This choice is motivated by Henderson *et alias* [71] and has been made to facilitate investigations of DDPG algorithm learning performance with respect to changes in exploration.

4.1.3 Software Implementation and Testing

Software implementations used for experiments will be developed using an object oriented approach. This design choice has been made to provide a cleaner and more modular implementation, allowing for easier modification of model elements and greater experimental flexibility. The principal elements of the software developed for experimentation are detailed in Table 4.1.

Table 4.1: Overview of software elements and their intended operation for simulation experiments

Software Element	Description
`Main`	The main function, executed from the terminal, instantiates `Environment`, `Agent`, and `Demand` objects that are passed to a `Training Algorithm` function call.
`Environment`	An object oriented implementation of the two area power system model in the temporal domain.
`Agent`	An object oriented implementation of the DDPG controller. When instantiated, an instance of the object `Model` is created and stored as an `Agent` object variable.
`Demand`	An object oriented implementation of a demand signal that is fed to the environment in order to perturb the system from steady state. The `Demand` implementation will provide for either Heaviside step function or stochasitic system perturbation.
`Model`	An oject oriented implementation of the neural network model instantiated when an `Agent` object is created.
`Training Algorithm`	A function that takes `Environment`, `Agent`, and `Demand` object arguments and initiates the DDPG training algorithm for a specified number of episodes.

Environment models, controllers, neural networks, and training algorithms will be implemented in the Python programming language. This design choice is motivated by the understanding that Python is one of the most widely used languages for machine learning applications and research prototyping for neural networks. Moreover, the Python programming language is open source allowing minimising of cost burdens compared to languages like Matlab. Finally, there are a number of well developed supporting libraries for scientific computing and neural network implementations. This research will make use of the following:

- `Scipy` is a Python library used for scientific computing. The library contains well

developed numerical computation schemes and algorithms for the simulation of systems of ordinary differential equations.

- `PyTorch` is a Python based machine learning framework that is used extensively for neural network research prototyping. One of the main benefits of this library is that it provides access to graphical processing hardware which accelerates the neural network training process.

- `Gym` is a Python toolkit developed by OpenAI for developing reinforcement learning algorithms.

Developed models for the environment, PI controller will be tested and verified against known results from the literature review to provide assurance of correct implementation. Implementations of neural networks and DDPG algorithms will be tested against known test problems to provide reasonable assurance of correct implementation.

An overview showing the hierarchy of software operations is outlined in Figure 4.1.

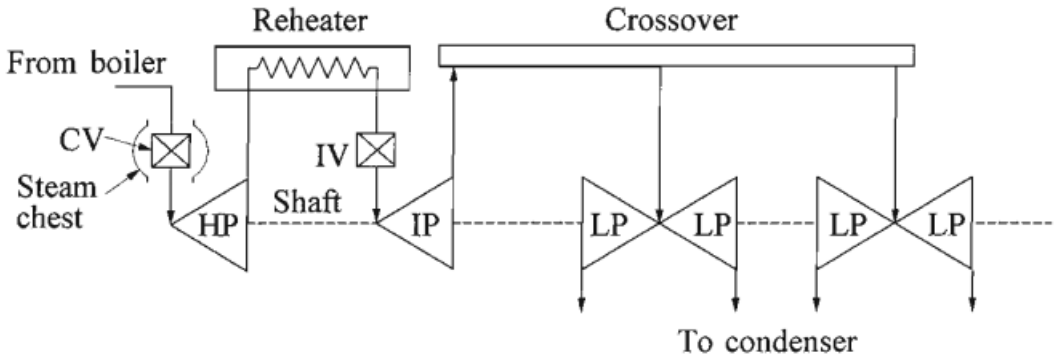

Figure 4.1: Execution of the main function creates `Env`, `Agent`, and `Demand` objects and passes them to a `ddpg_train` function call

4.2 Simulation Experiments

4.2.1 Overview

As discussed in §2.4.2, DDPG performance has been observed to suffer from instability and variance during training. This can impact the agent's ability to learn useful control policies. Some of the main causes of variability in agent performance are due to neural network architecture, choice of activation function, exploratory noise processes, agent experience quality, and agent experience variability. The simulation experiment aims are twofold. Firstly, identify parameter settings and network architectures for which a DDPG

agent can learn a frequency control policy comparable to an optimally tuned proportional-integral (PI) controller for a two area power system. Secondly, hyperparameter settings will be identified to ensure agent learning is stable and fast.

To achieve these aims, a series of experiments will be undertaken in which a neural network will be trained using DDPG to eliminate frequency change in a two area power system subjected to load demand changes. Each experiment will be run for an identical fixed number of training episodes, with each episode lasting for a 30 second duration. Each experiment will modify the neural network architecture or the DDPG algorithm to find the most effective control policy that demonstrates stable and fast learning. Fixed random seeds will be used to ensure each experiment sees identical training scenarios, and to ensure results can be replicated. Table 4.2 provides an overview of proposed experiments. Agent performance will be evaluated according to the criteria outlined in section 4.2.2.

Table 4.2: An overview of the experiments that will undertaken to achieve the best neural network performance for load frequency control of a two area power system.

Name	Description
Baseline	An identical neural network architecture used in Lillicrap *et alias*, with hidden layers of (400, 300) will be used to form a basis comparisons against subsequent experiments.
Neural Network Architecture	A neural network architecture that has smaller number of nodes in hidden layers (64, 64) will be used to determine if a smaller network is more effective.
Activation Functions	LReLU activation functions will be used for neural network hidden layers to determine if they are more effective.
OU Noise Process	Exploratory OU noise process parameters, σ and θ, will be modified to determine the impact on agent learning.
Prioritised Experience Replay	Priority metric added to experience replay buffer allowing useful experience to be visited more often.
Expert Learner	PID controller experience (the expert) will be introduced to the experience replay buffer with half of all experience from PID interaction with the environment.
Stochastic Demand	A stochastic load demand profile will be used to perturb the system to provide the agent with more variability in training scenarios.

4.2.2 Measuring agent performance

MAKE SURE TO PROVIDE CLEAR METHODOLOGY ON HOW THE THE AGENTS TRAINING PROGRESS WILL BE MEASURED AND COMPARED

MAKES SURE THAT IT IS CLEAR ON HOW THE CONTROLLER PERFORMANCE WILL BE MEASURED AND THE COMPARISONS THAT WILL BE MADE FOR EACH OF THE EXPERIMENTS.

Chapter 5

Development, Implementation, and Testing

Model development is focused on obtaining a mathematical expression for modelling the two area power system and the PI controller, in addition to specifying the neural network architectures for experiments. Implementation primarily focuses on the DDPG training algorithm along with Python class implementations for the environment, PI controllers, and neural networks.

PROVIDE AND OVERVIEW OF THE SOFTWARE THAT HAS BEEN DEVELOPED - COMMUNICATE THIS IDEA BY CREATING A TREE LIKE STRUCTURE

5.1 Environment Model

The two area power system model described in §2.1.3, and shown in Figure 2.14, was converted from the frequency domain to the temporal domain. This provides the reinforcement learning architectures, outlined in §2.2, with the ability to export control signals to the power system at each time step. This approach is common practice when developing environments for reinforcement learning [72]. Additionally, frequency domain simulation techniques, such as Laplacian transforms, have strict initial condition assumptions [53], which limit the application of this technology in real world applications.

Higher order ordinary differential equations and systems involving higher order ordinary differential equations provide a more compact system representation; however, to accommodate numerical analysis schemes, such as Runge-Kutta, the environment system was expressed as a system of first-order linear ordinary differential equations.

Suppose variables are assigned to the power system according to Figure 5.1. The first

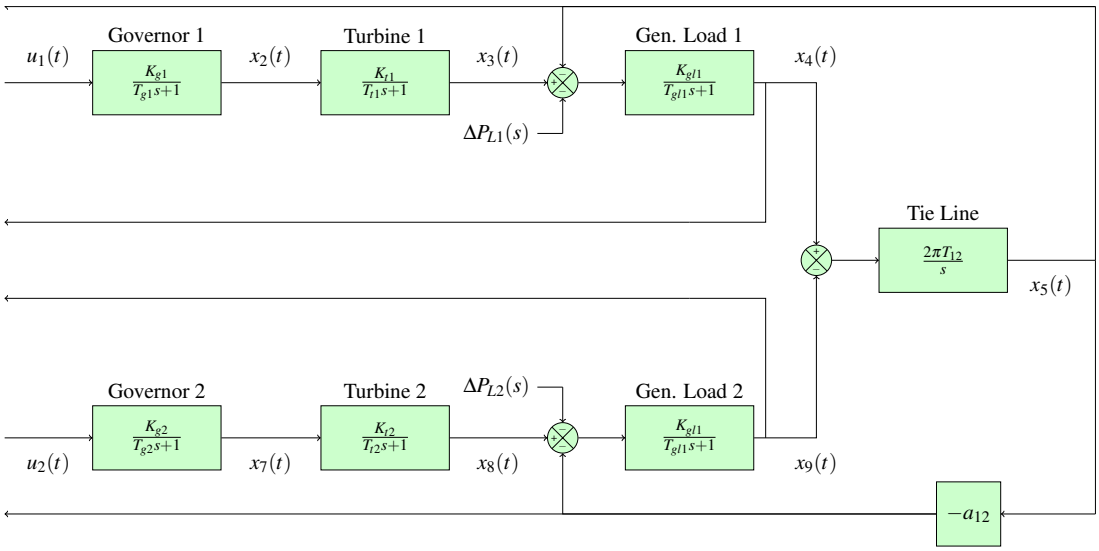

Figure 5.1: Variable assignment for a two area power system environment to model in the temporal domain

order system of linear ordinary differential equations for area 1 is:

$$\dot{x}_2(t) = \frac{1}{T_{sg_1}}\left(K_{sg_1}u_1(t) - x_2(t)\right) \tag{5.1}$$

$$\dot{x}_3(t) = \frac{1}{T_{t_1}}\left(K_{t_1}x_2(t) - x_3(t)\right) \tag{5.2}$$

$$\dot{x}_4(t) = \frac{1}{T_{gl_1}}\left(K_{gl_1}\left(x_3(t) - x_5(t) - \Delta p_{L1}(t)\right) - x_4(t)\right) \tag{5.3}$$

$$\dot{x}_5(t) = 2\pi T_{12}\left(x_4(t) - x_9(t)\right) \tag{5.4}$$

$$\dot{x}_7(t) = \frac{1}{T_{sg_2}}\left(K_{sg_2}u_2(t) - x_7(t)\right) \tag{5.5}$$

$$\dot{x}_8(t) = \frac{1}{T_{t_2}}\left(K_{t_2}x_7(t) - x_8(t)\right) \tag{5.6}$$

$$\dot{x}_9(t) = \frac{1}{T_{gl_2}}\left(K_{gl_2}\left(x_8(t) - x_5(t) - \Delta p_{L2}(t)\right) - x_9(t)\right) \tag{5.7}$$

A full derivation of the first order linear system described by Equations 5.1 to 5.7 is described in Appendix C.1. The equations were implemented as a method `int_power_system_sim` in a Python class `TwoAreaPowerSystemEnv`. Implementation for `int_power_system_sim` is detailed in Appendix D.1.

ADD THE PARAMETER SELECTION FROM LITERATURE REVIEW.

ADD TESTS TO ENSURE THAT MODEL SPECIFICATION IS OKAY - USE NO CONTROL FROM FREQUENCY DOMAIN AND FROM TEMPORAL PYTHON MODEL.

Table 5.1: Environment and controller parameters used for preliminary investigation experiments.

Description	Parameter	Value
Speed Governor Gain	K_{sg_1}, K_{sg_2}	1
Speed Governor Time Constant	T_{sg_1}, T_{sg_2}	0.08
Turbine Gain	K_{t_1}, K_{t_2}	1
Turbine Time Constant	T_{t_1}, T_{t_2}	0.3
Generator Load Gain	K_{gl_1}, K_{gl_2}	120
Generator Load Time Constant	T_{gl_1}, T_{gl_2}	20
Tieline	T_{12}	0.1
Proportional Gain	R_1, R_2	2.4
Integral Gain	K_{i_1}, K_{i_2},	-0.671
	b_1, b_2	0.425

5.2 Classical PI Controller Model

The proportional integral (PI) controller model described in §2.1.4, and shown in Figure 2.14, was converted from the frequency domain to the temporal domain to ensure compatibility with the environment model.

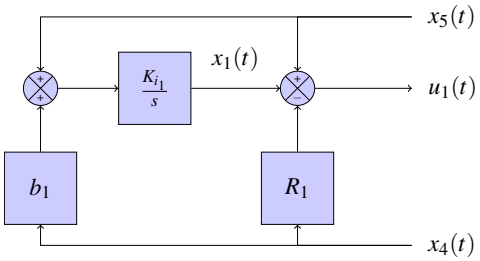

Figure 5.2: Variable assignment for the PI controller for area 1 in order to model in the temporal domain

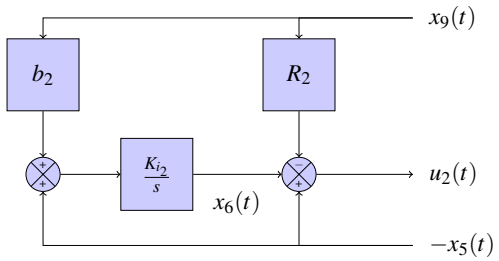

Figure 5.3: Variable assignment for the PI controller for area 2 in order to model in the temporal domain

Suppose variables are assigned to the controller for area 1 and the controller for area 2 according to Figures 5.2 and 5.3, respectively. The first order system of linear differential equations for the PI controller are:

$$\dot{x}_1(t) = b_1 \Delta f_1(t) + x_5(t) \tag{5.8}$$

$$\dot{x}_6(t) = b_2 \Delta f_2(t) - x_5(t) \tag{5.9}$$

Note that once the PI controller has stepped forward in time during simulation the actual control signals exported from the controller are $u_1(t)$ and $u_2(t)$. These are expressed as:

$$u_1(t) = x_1(t) + x_5(t) - R_1 x_4(t) \tag{5.10}$$

$$u_2(t) = x_6(t) - x_5(t) - R_2 x_9(t) \tag{5.11}$$

A full derivation of the first order linear system described by equations 5.8 and 5.9 is described in Appendix C.2. The system of equations is implemented as a method `int_control_system_sim` in a Python class `ClassicalPiController`. Implementation for `int_control_system_sim` is detailed in Appendix D.2.

5.3 Neural Network Controller Model

Deep deterministic policy gradient (DDPG), described in §2.4.2, is used to train neural networks to perform control actions given a state observation. DDPG uses actor and critic networks, both of which also have target networks. The implication is that a total of four neural networks are required for a single DDPG controller instance. To accommodate this requirement, two Python classes were created: one for the actor called `Actor`, and another for the critic called `Critic`. Implementations for the `Actor` and `Critic` classes can be found in Appendices D.3 and D.4, respectively. Both actor and critic networks were built using an input layer, two hidden layers and an output layer — an identical structure to experiments conducted by Lillicrap *et al.* [33]. Neural network hidden layers used rectified linear units (ReLU) for activation functions. The final output layer of the actor network used a tanh activation function to bound the actions.

5.4 DDPG Algorithm

The DDPG algorithm was implemented using a Python class `DdpgController`, and a function called `ddpg_train`.

When an instance of the `DdpgController` class is created a number of critical tasks are performed. These include initialisation of neural networks in memory for the actor and critic, and their associated target networks, declaration of DDPG hyperparameters discussed in §2.4.2, initialisation of a replay buffer for storing agent experience, and the initialisation of an Ornstein–Uhlenbeck process to add exploratory noise to action signals. Additionally, the `DdpgController` class defines methods used for model training. A description of key methods is provided in Table 5.2. The `DdpgController` class implementation can be found in Appendix D.7.

Actor and critic neural networks are trained using the DDPG algorithm shown in listing 2.4.2.

The function `ddpg_train` takes environment and agent class instances as inputs. The function runs training for a specified number of episodes. For each episode, `ddpg_train` takes time steps of 0.01 sec until the episode triggers a termination condition. During each time step, `ddpg_train` will obtain an action from the agent given the current state, obtain the power demand signal for the given time step, and step the environment forward using

Table 5.2: Description of key methods for the `DdpgController` class

Method	Description
`step`	Stores the current experience tuple, (s_t, a_t, r_t, S_{t+1}), in the experience replay buffer, and calls the method `learn`
`act`	Takes a state as input and uses this for a call to the neural network method `forward` which returns an action
`learn`	Performs gradient descent, using backpropagation, on the actor and critic loss functions to adjust weights for each respective network using a set of random uniformly sampled experiences from the experience replay buffer

the agent's action and power demanded. The function the makes a call to the agent method `step`, which stores the experience in the replay buffer and trains the neural network.

Chapter 6

Simulation Experiments

Simulation experiments, outlined in section 4.2 were undertaken to determine the suitability of using a neural network for load frequency control of a two area power system. This chapter details common aspects across each experimental set-up, explains the experimental process for each simulation experiment, and documents the results.

6.1 General Experimental Setup

Experiments consisted of two main phases: training and testing.

6.1.1 Training

At the beginning of each experiment a new instance of a DDPG agent was initialised, and the DDPG agent replay buffers cleared. Each experiment used the same episode scenario, with the system and agent response simulated for a total of 30 sec, after which the episode was terminated. In order to simulate a power system perturbation, a $\pm$0.01pu step change in the power demand for area 1 was introduced at a random time between the 0 and 30 sec mark. An example of a +0.01pu step change occurring at the 1 sec mark is shown in Figure 6.1. Note that this perturbation type was not used for the final experiment.

From initialisation, the simulation was incrementally stepped forward by 0.01 sec for a total of 3000 steps. At each time step the DDPG agent was trained using experience stored in the replay buffer from current and previous system interactions, for a given experiment. Experiments were allowed to run for a total of 10000 episodes.

DDPG training algorithm hyperparameters were held constant for each experiment, as per the values used originally by Lillicrap *et alias* [33]. Hyperparameter values used for the experiments described in this chapter are documented in Table 6.1.

Each experiment, detailed in the remaining sections of this chapter, modified a single variable of the agent construction, holding other variables constant. and cumulative

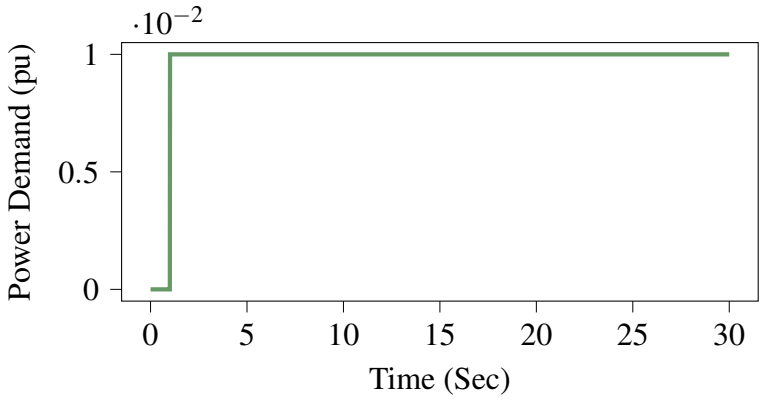

Figure 6.1: At the 1 sec mark the system experiences a step load change in the power demand in Area 1, and the simulation continues for 30 sec thereafter.

Table 6.1: DDPG hyperparameters used for preliminary investigation experiments.

Hyperparameter	**Value**	**Hyperparameter**	**Value**
Buffer Size	1×10^6	Batch Size	256
Gamma (γ)	0.99	Tau (τ)	1×10^{-3}
Actor Learning Rate ($\alpha_{\texttt{actor}}$)	1×10^{-4}	Critic Learning Rate ($\alpha_{\texttt{critic}}$)	3×10^{-4}
Weight Decay	0.00		

reward for each episode. for each power system area were captured and reported as experimental results.

6.1.2 Testing

A series of plots were created for each experiment. These include the following:

- **Frequency:** a timeseries plot showing an evolution of power system frequency under neural network control, for a given power system area, over the 30 sec simulation. Two plots will be created for each experiment — one for each power system area. Note that system frequency under optimal PI control will also be included for comparison.

- **Control signal:** a timeseries plot showing an evolution of the control action issued by the neural network over the 30 sec simulation. Two plots will be created for each experiment — one for each experiment. Note that control signals from an optimal PI controller will also be included for comparison.

6.2 Baseline Experiment

Actor network architectures consisted of an input layer (7 perceptrons), a single hidden layer (256 perceptrons), and an output layer (2 perceptrons). Hidden layers used ReLU activation functions, and the output layer used a tanh activation function. The network architectures of the critic consisted of an input layer (7 perceptrons), three hidden layers (256, 256, and 128 perceptrons), and an output layer (1 nodes). Hidden layers used LReLU activation functions, and the output layer used no activation function.

Table 6.2: An overview of actor and critic neural network architectures used in experiments undertaken by Lillicrap *et alias*

Layer	Actor		Critic	
	Activation Function	**Number of Perceptrons**	**Activation Function**	**Number of Perceptrons**
Input Layer	None	3	None	3
Hidden Layer 1	ReLU	400	ReLU	400
Hidden Layer 2	ReLU	300	ReLU	300
Output Layer	tanh	2	None	2

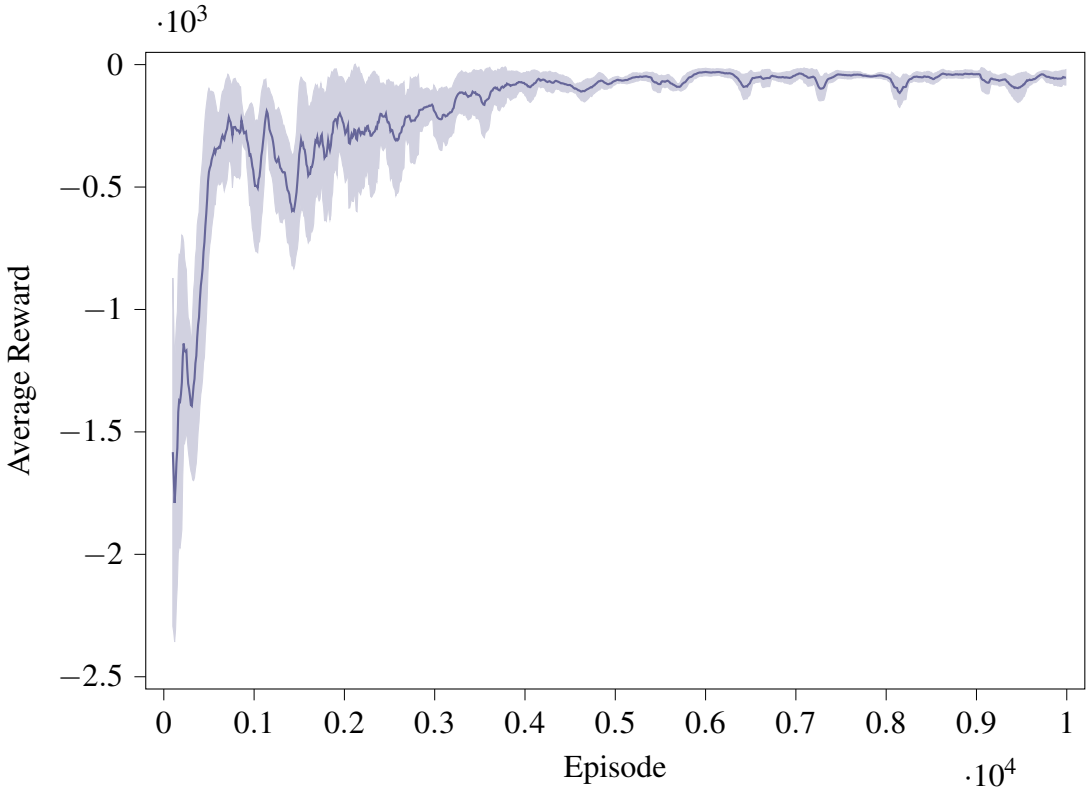

Figure 6.2: text

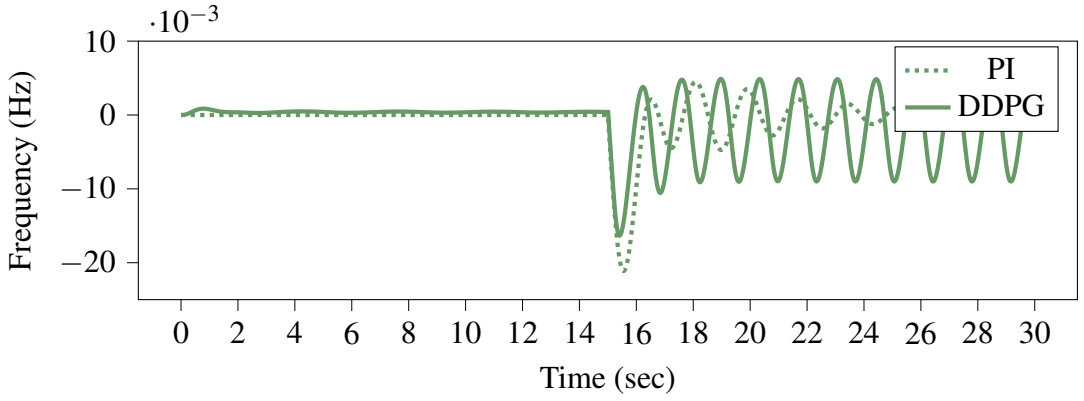

Figure 6.3: text

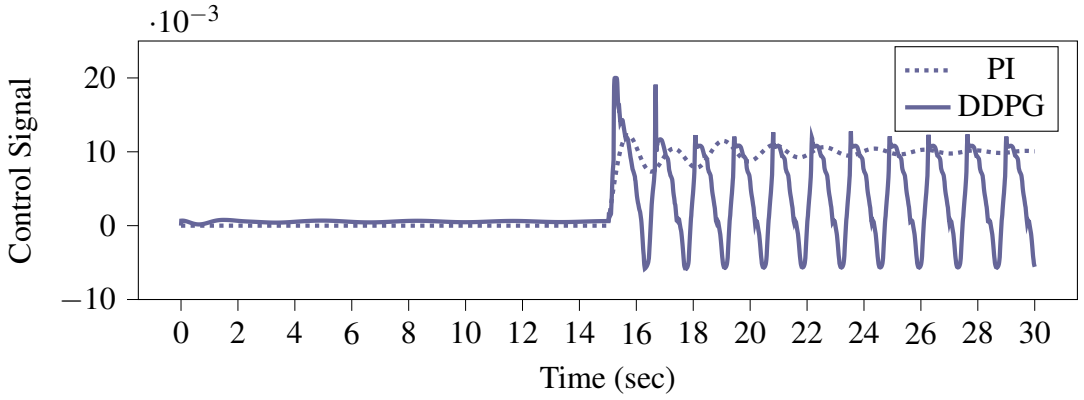

Figure 6.4: text

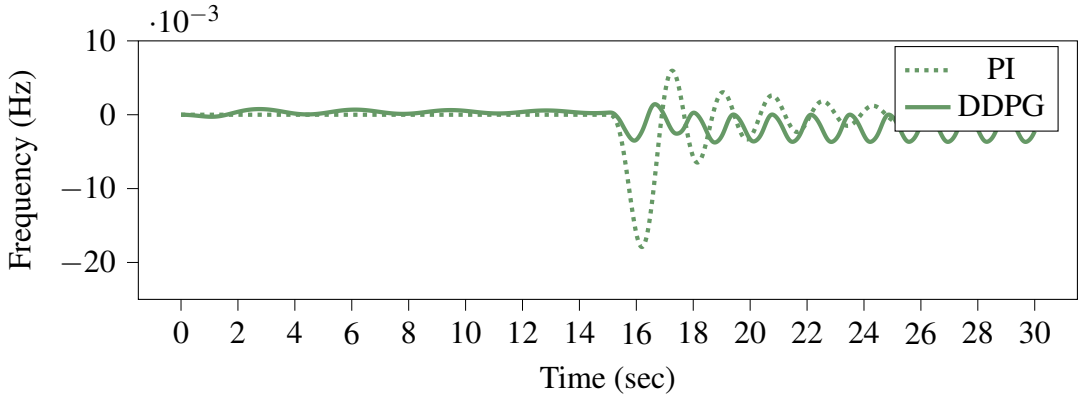

Figure 6.5: text

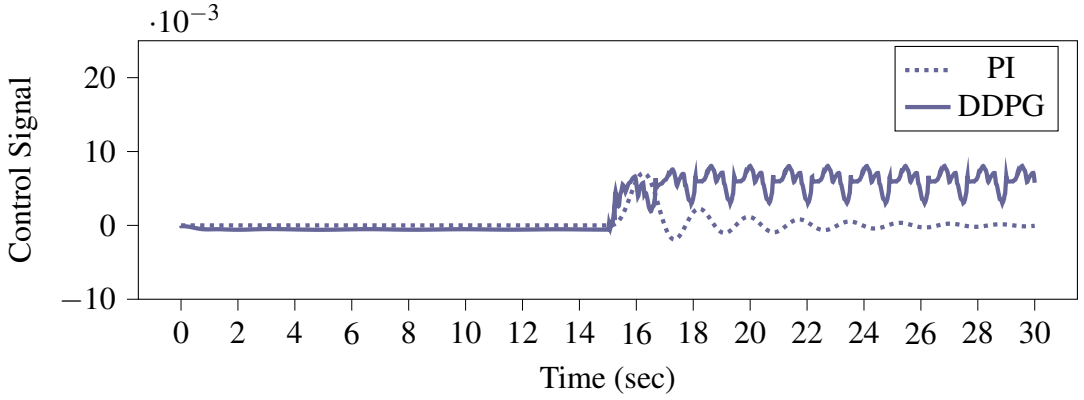

Figure 6.6: text

6.3 Neural Network Architecture Experiments

Both actor and critic architectures contain 121800 weight parameters. The details of the actor and critic networks are shown in Table 6.3.

Table 6.3: An overview of actor and critic neural network architectures used in experiments undertaken by Lillicrap *et alias*

Layer	Actor		Critic	
	Activation Function	**Number of Perceptrons**	**Activation Function**	**Number of Perceptrons**
Input Layer	None	3	None	3
Hidden Layer 1	ReLU	400	ReLU	400
Hidden Layer 2	ReLU	300	ReLU	300
Output Layer	tanh	2	None	2

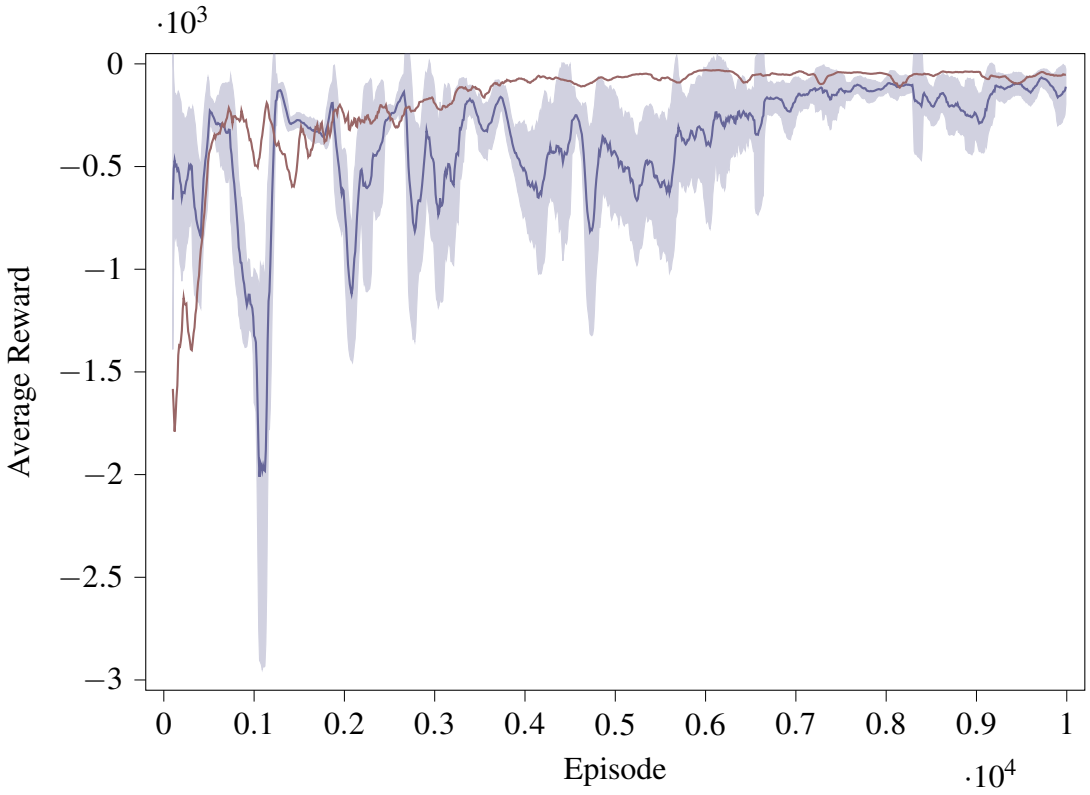

Figure 6.7: text

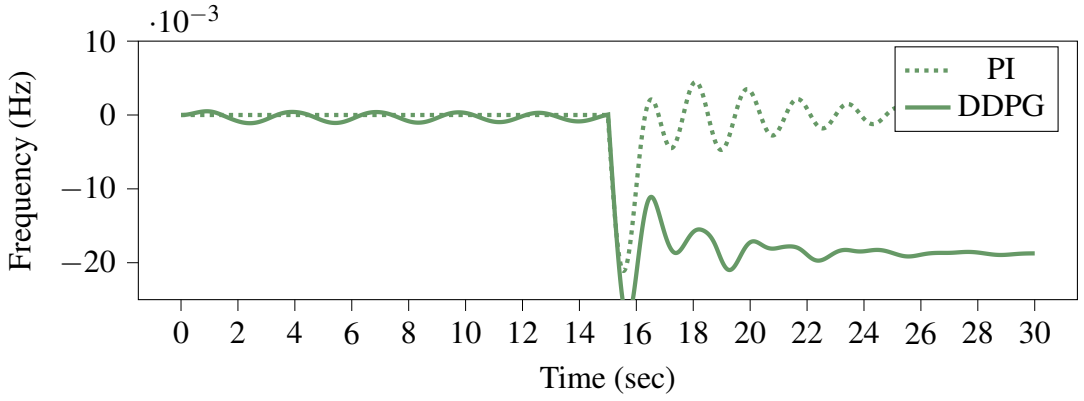

Figure 6.8: text

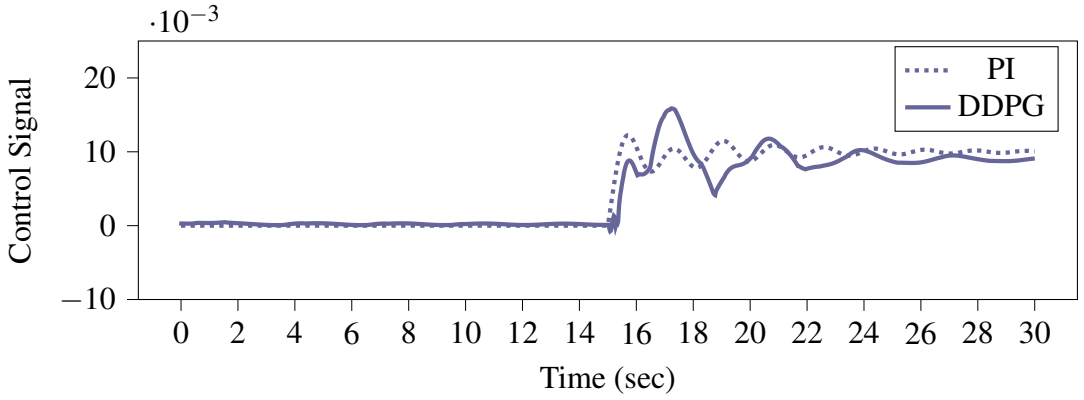

Figure 6.9: text

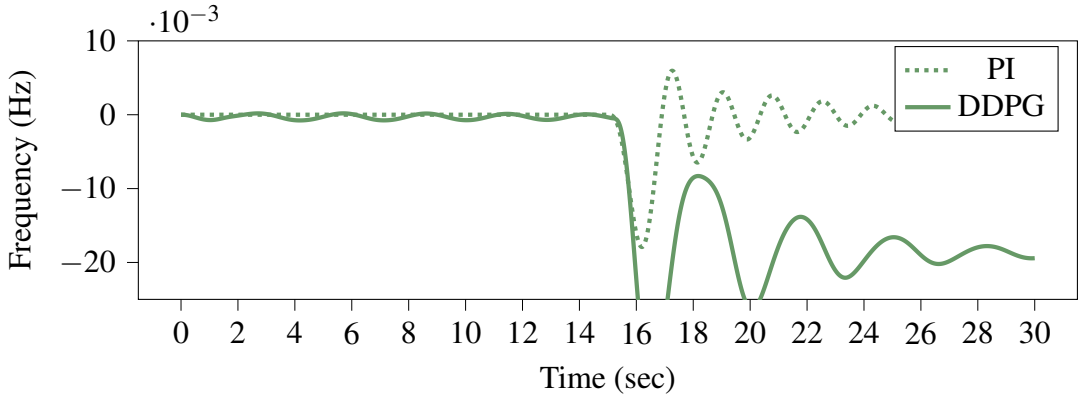

Figure 6.10: text

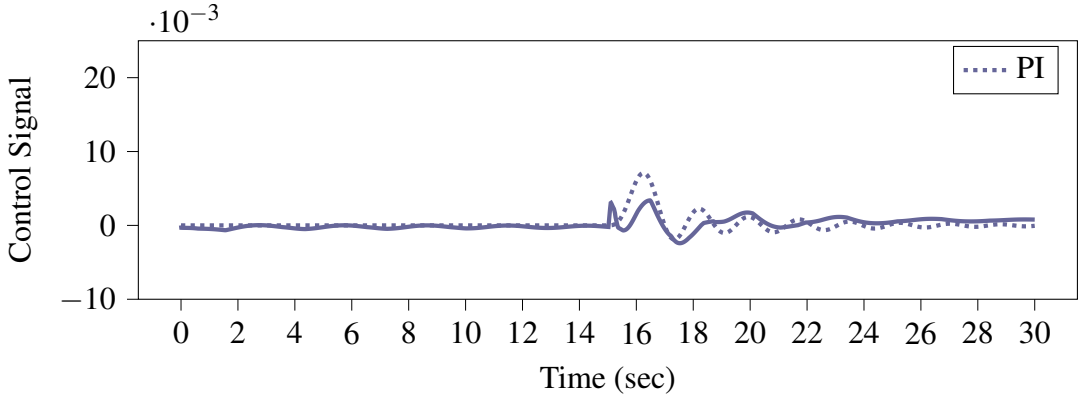

Figure 6.11: text

6.4 Activation Function Experiments

Table 6.4: An overview of actor and critic neural network architectures used in experiments undertaken by Lillicrap *et alias*

Layer	Actor		Critic	
	Activation Function	**Number of Perceptrons**	**Activation Function**	**Number of Perceptrons**
Input Layer	None	3	None	3
Hidden Layer 1	ReLU	400	ReLU	400
Hidden Layer 2	ReLU	300	ReLU	300
Output Layer	tanh	2	None	2

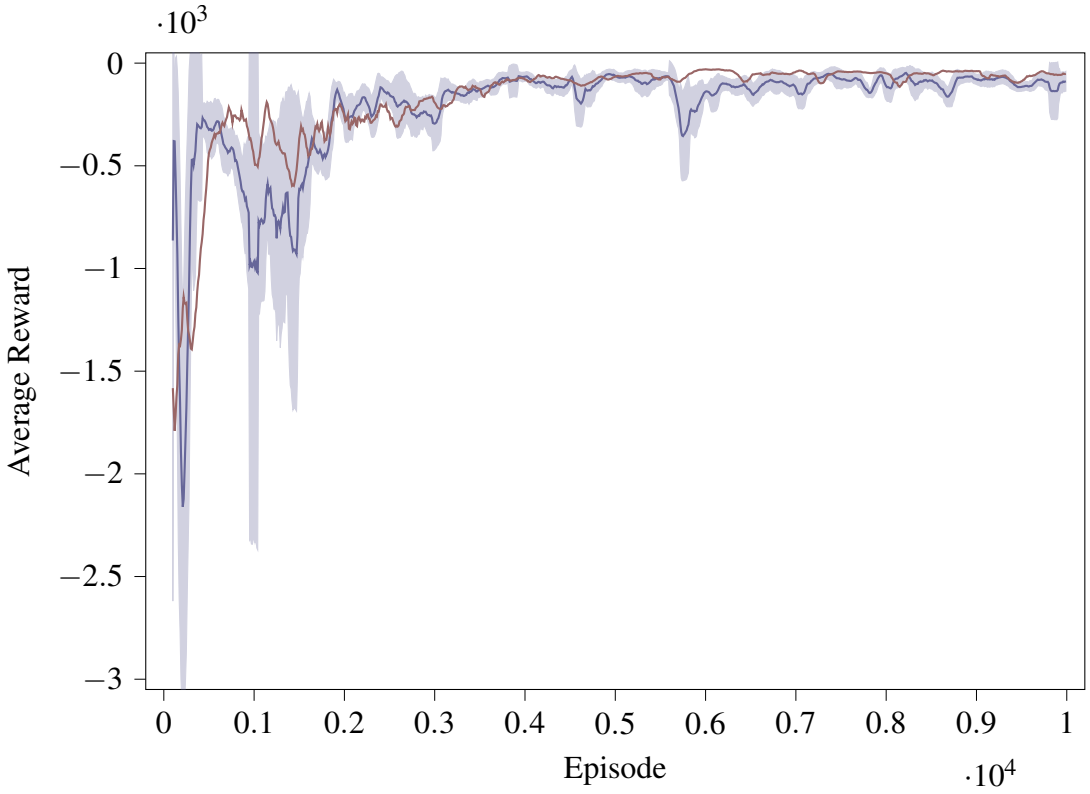

Figure 6.12: text

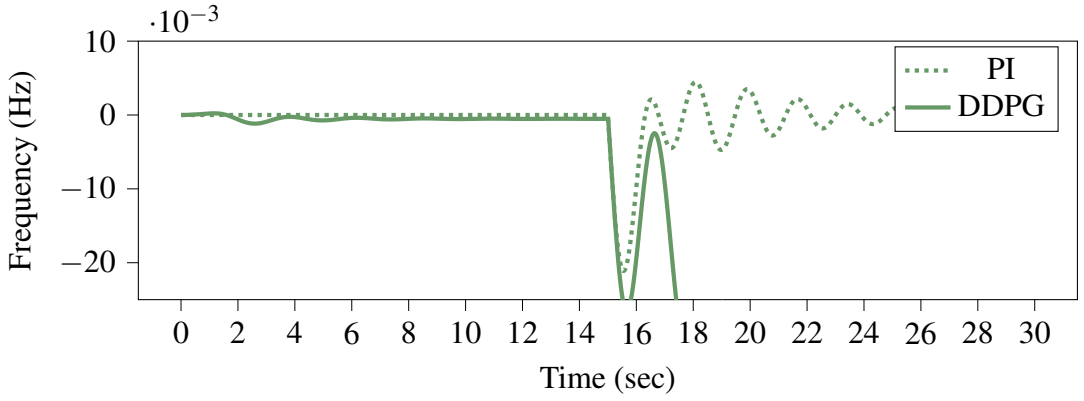

Figure 6.13: text

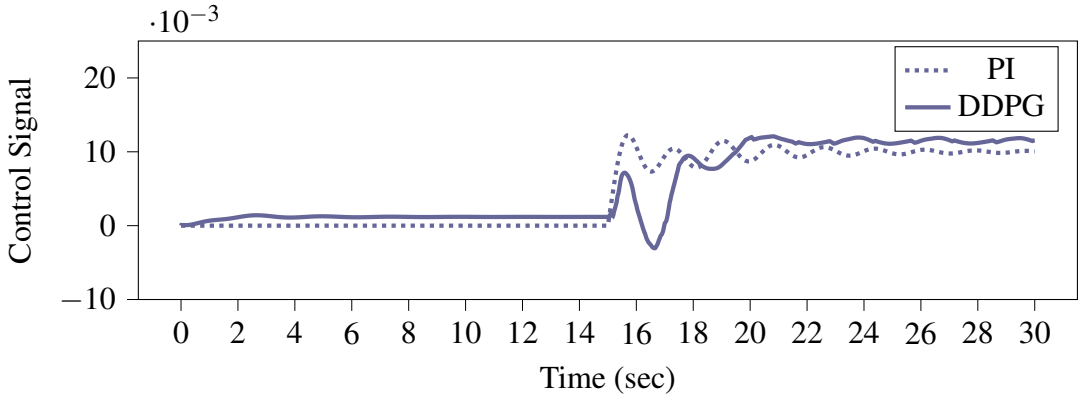

Figure 6.14: text

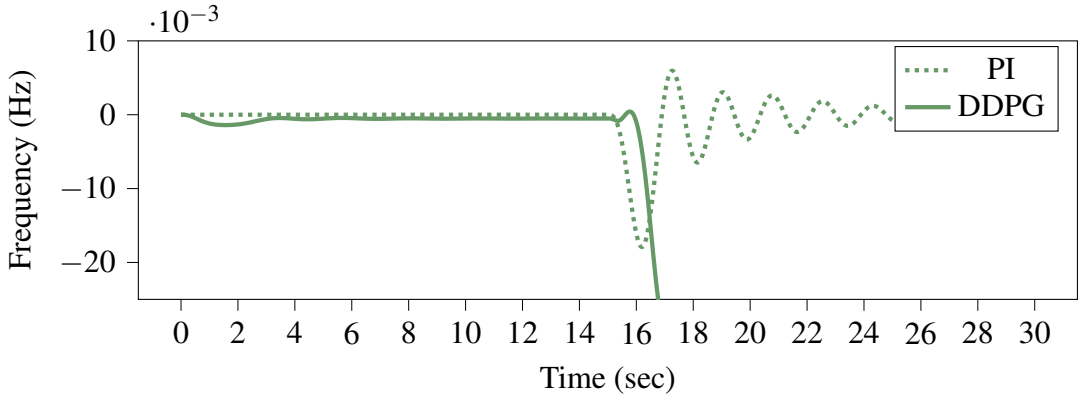

Figure 6.15: text

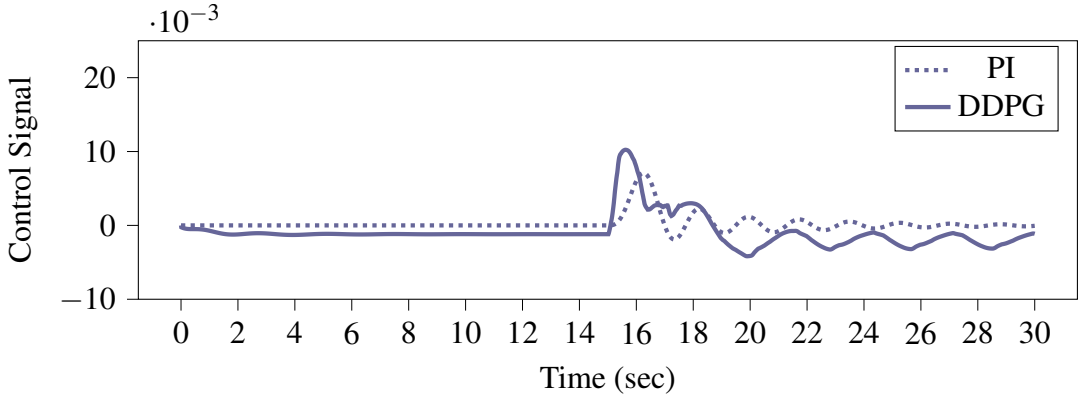

Figure 6.16: text

6.5 OU Noise Hyperparameter Experiments

Table 6.5: An overview of actor and critic neural network architectures used in experiments undertaken by Lillicrap *et alias*

Layer	Actor		Critic	
	Activation Function	**Number of Perceptrons**	**Activation Function**	**Number of Perceptrons**
Input Layer	None	3	None	3
Hidden Layer 1	ReLU	400	ReLU	400
Hidden Layer 2	ReLU	300	ReLU	300
Output Layer	tanh	2	None	2

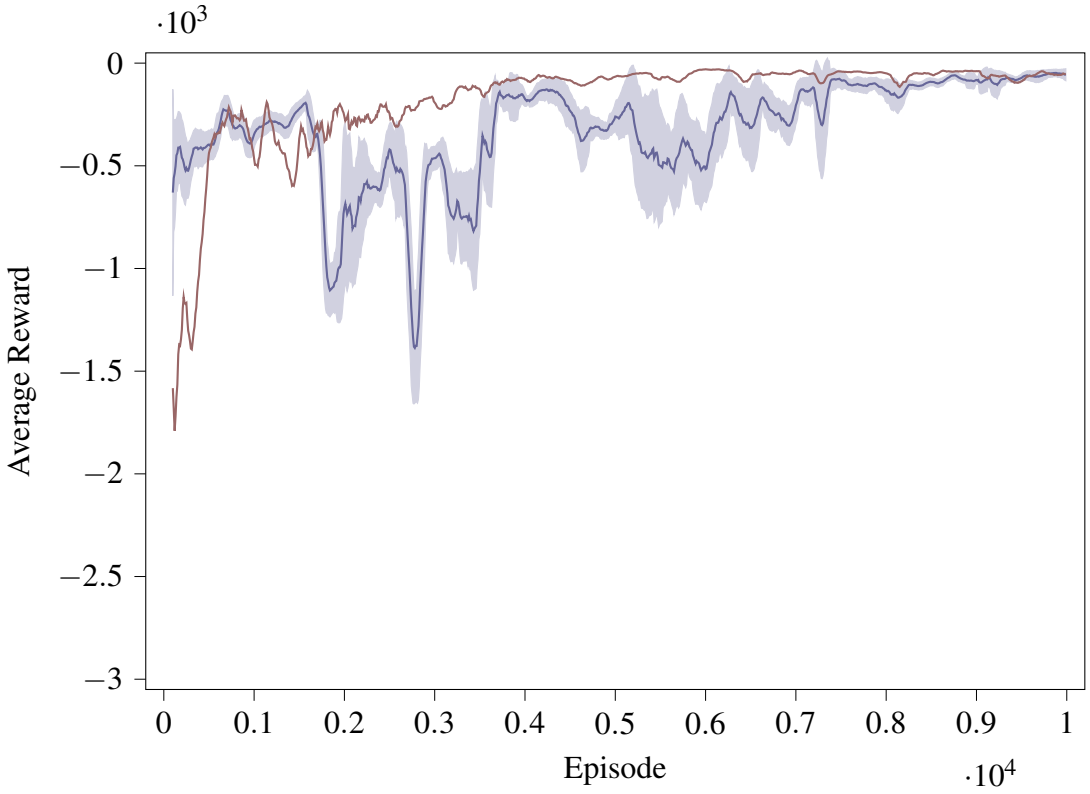

Figure 6.17: text

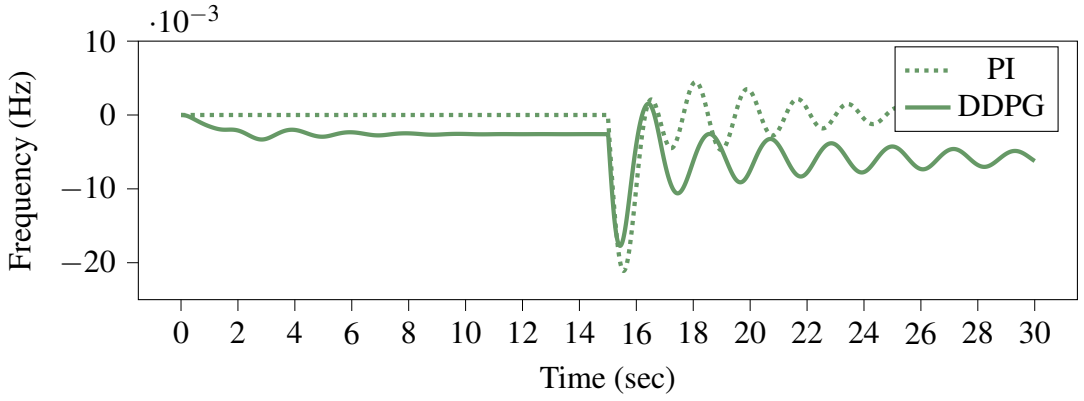

Figure 6.18: text

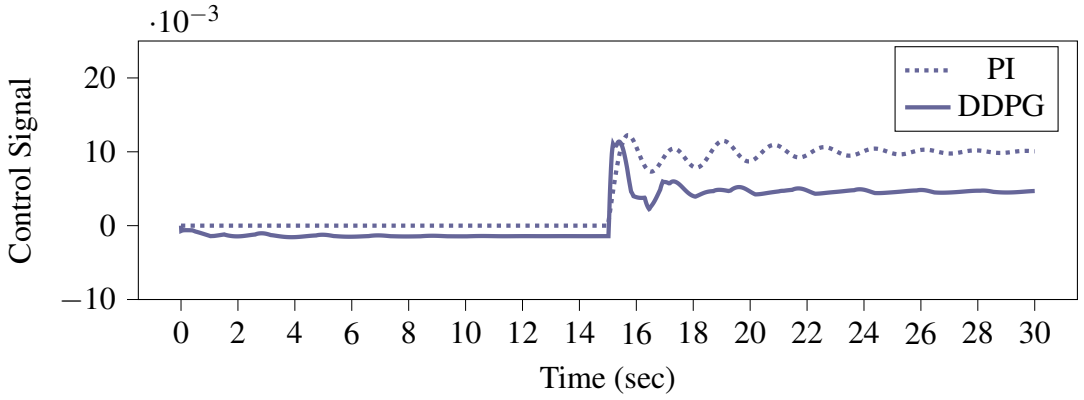

Figure 6.19: text

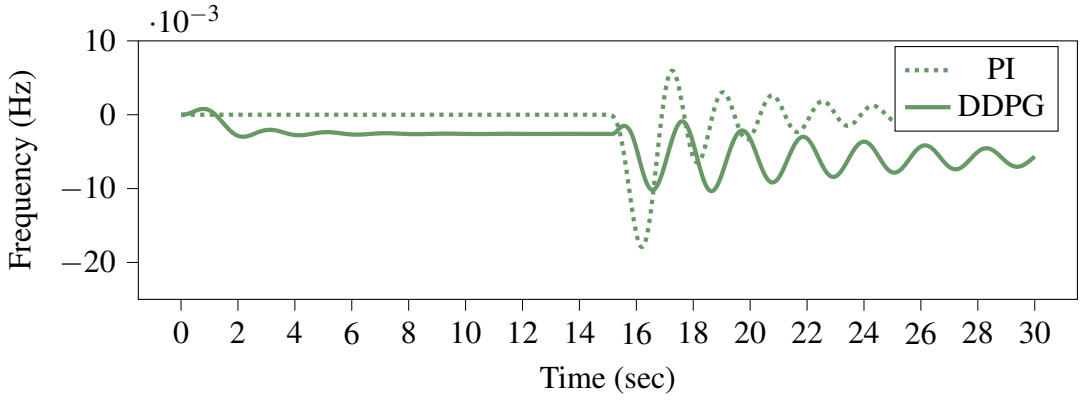

Figure 6.20: text

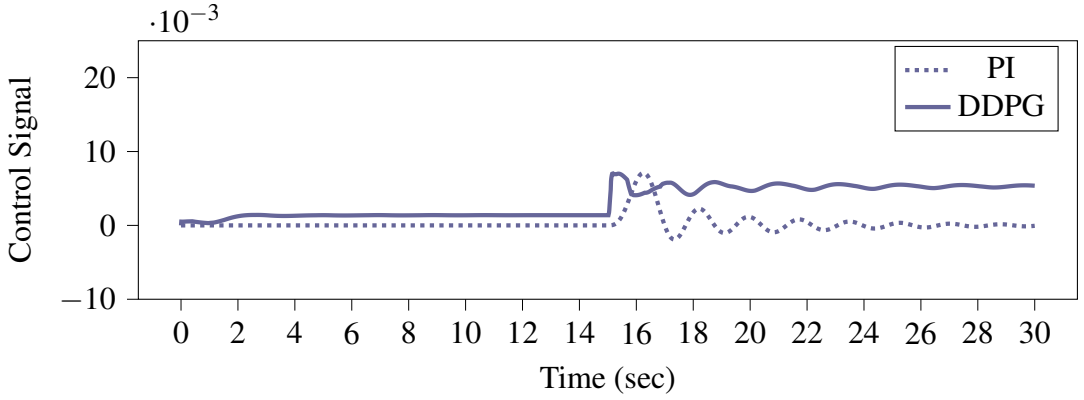

Figure 6.21: text

6.6 Priority Experience Replay Experiment

Table 6.6: An overview of actor and critic neural network architectures used in experiments undertaken by Lillicrap *et alias*

Layer	Actor		Critic	
	Activation Function	**Number of Perceptrons**	**Activation Function**	**Number of Perceptrons**
Input Layer	None	3	None	3
Hidden Layer 1	ReLU	400	ReLU	400
Hidden Layer 2	ReLU	300	ReLU	300
Output Layer	tanh	2	None	2

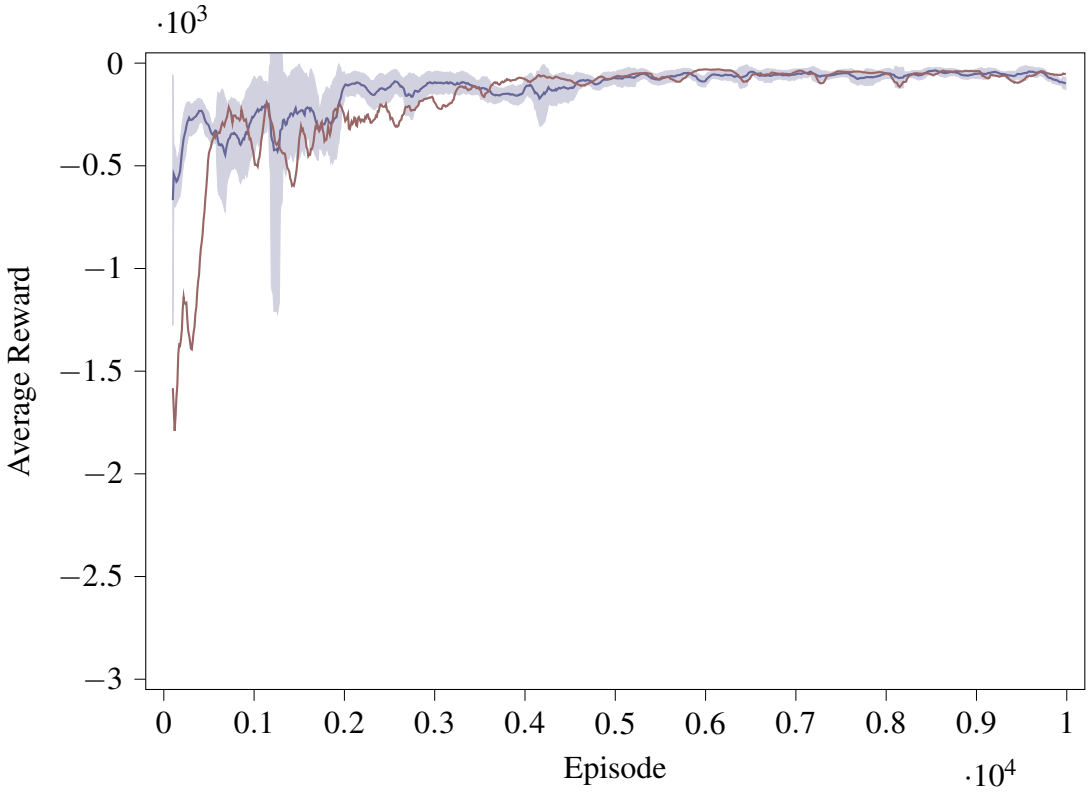

Figure 6.22: text

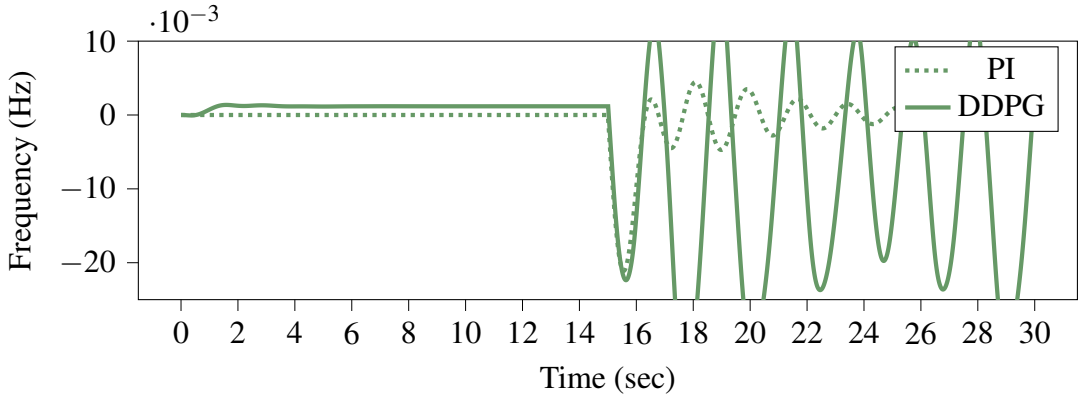

Figure 6.23: text

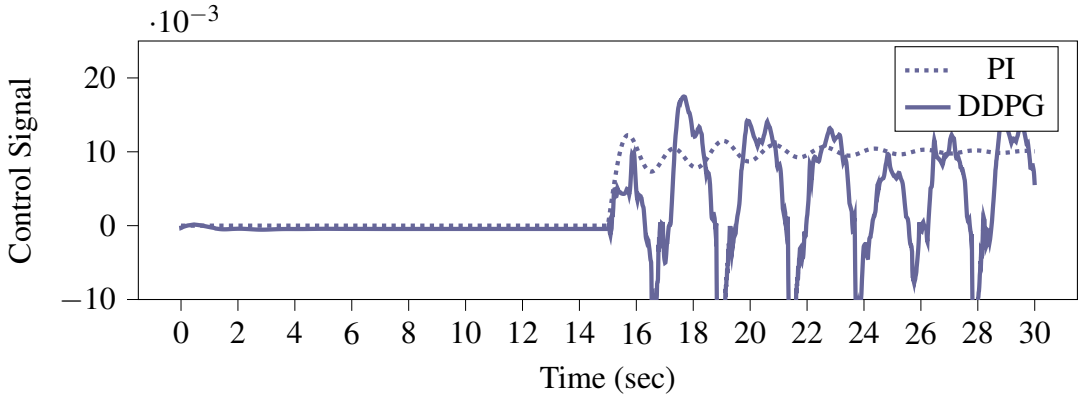

Figure 6.24: text

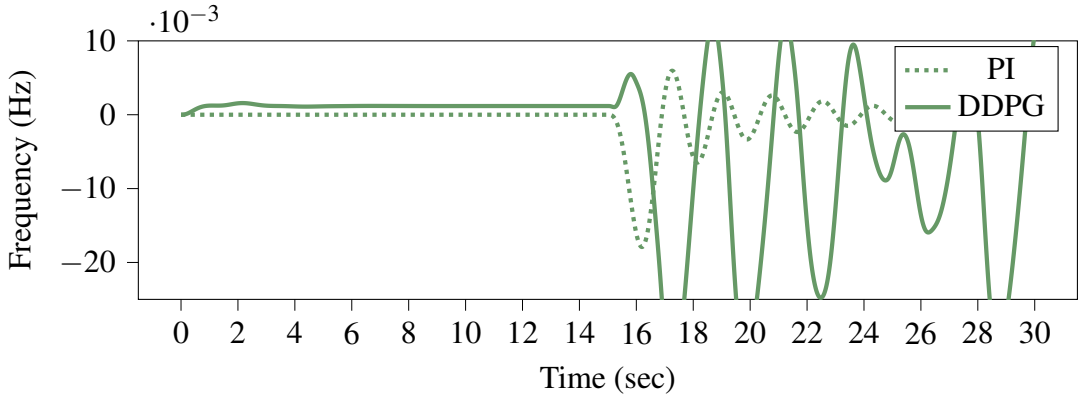

Figure 6.25: text

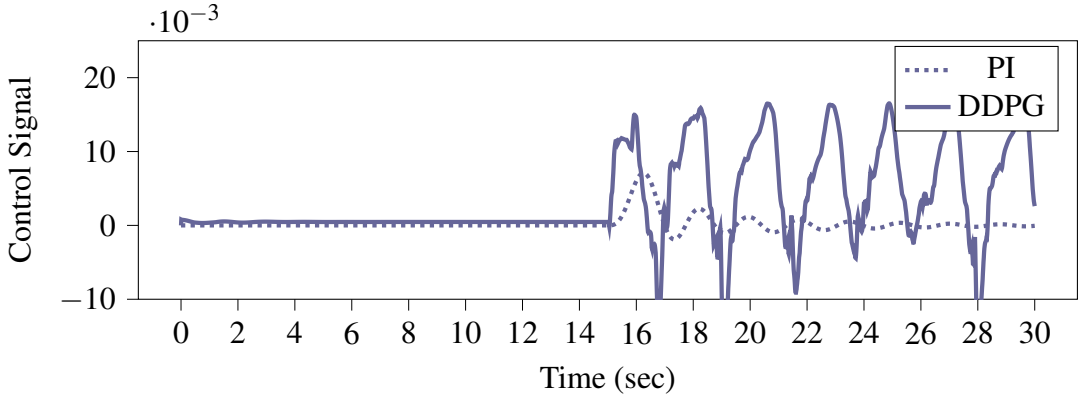

Figure 6.26: text

6.7 Expert Learner Experiment

Table 6.7: An overview of actor and critic neural network architectures used in experiments undertaken by Lillicrap *et alias*

Layer	Actor		Critic	
	Activation Function	**Number of Perceptrons**	**Activation Function**	**Number of Perceptrons**
Input Layer	None	3	None	3
Hidden Layer 1	ReLU	400	ReLU	400
Hidden Layer 2	ReLU	300	ReLU	300
Output Layer	tanh	2	None	2

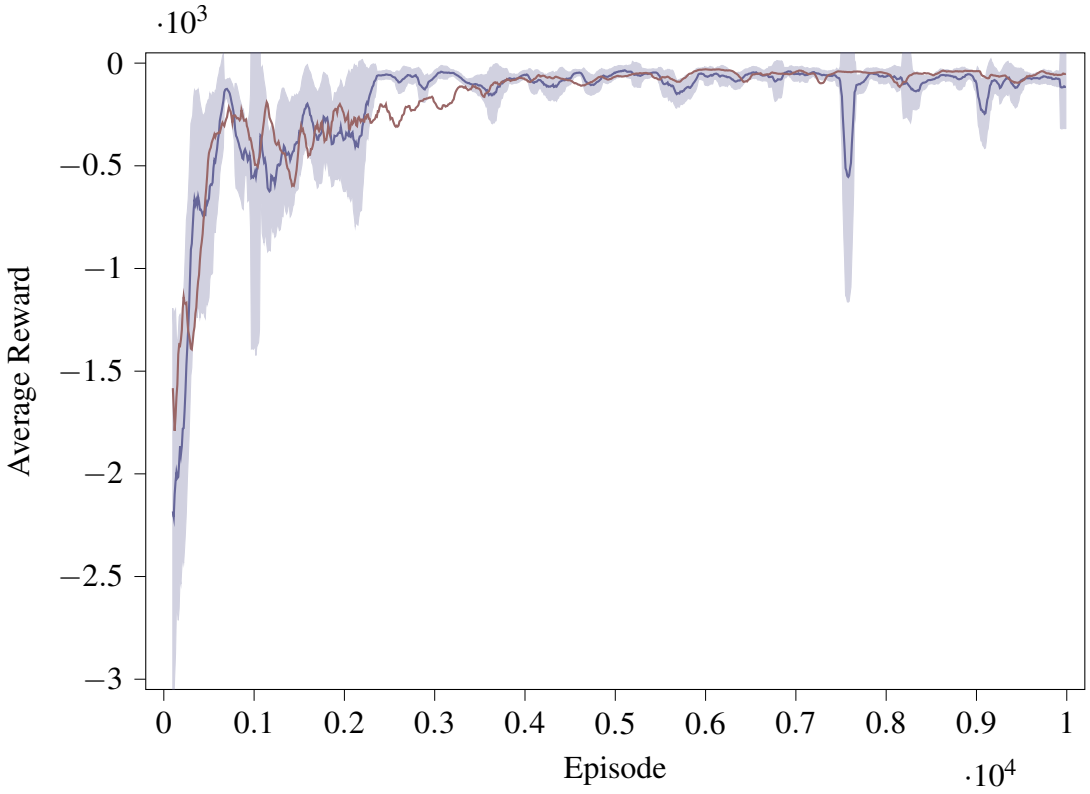

Figure 6.27: text

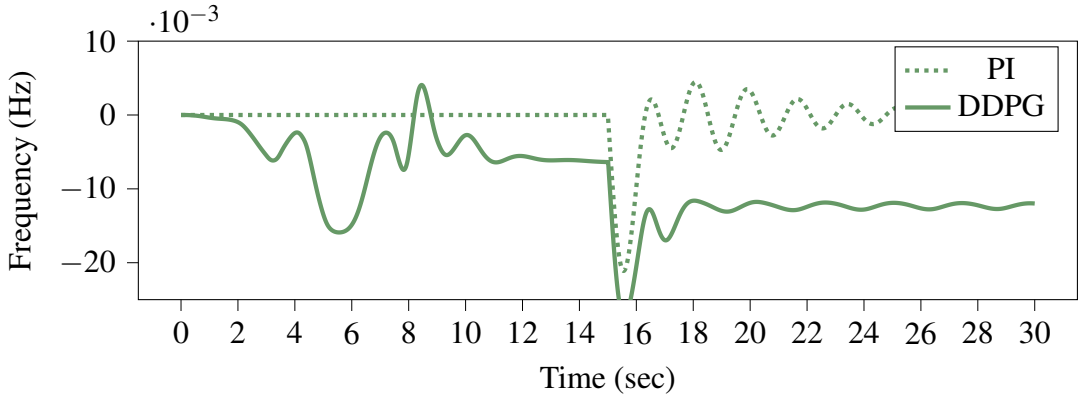

Figure 6.28: text

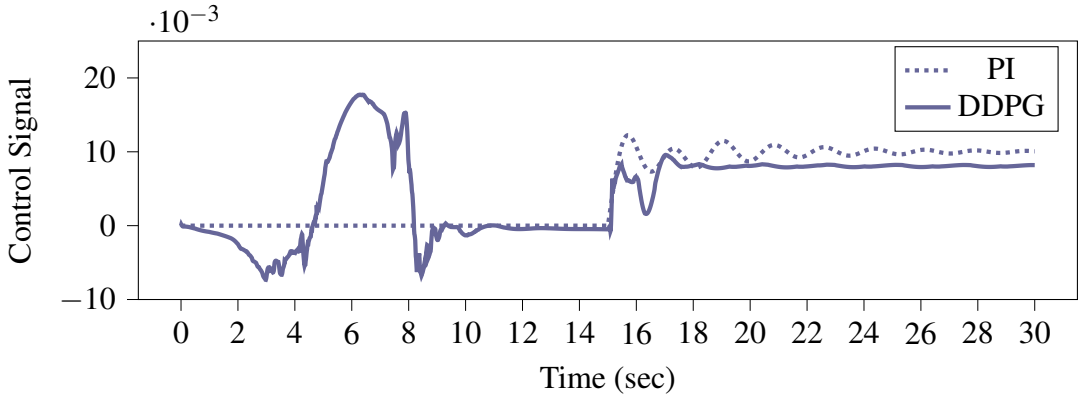

Figure 6.29: text

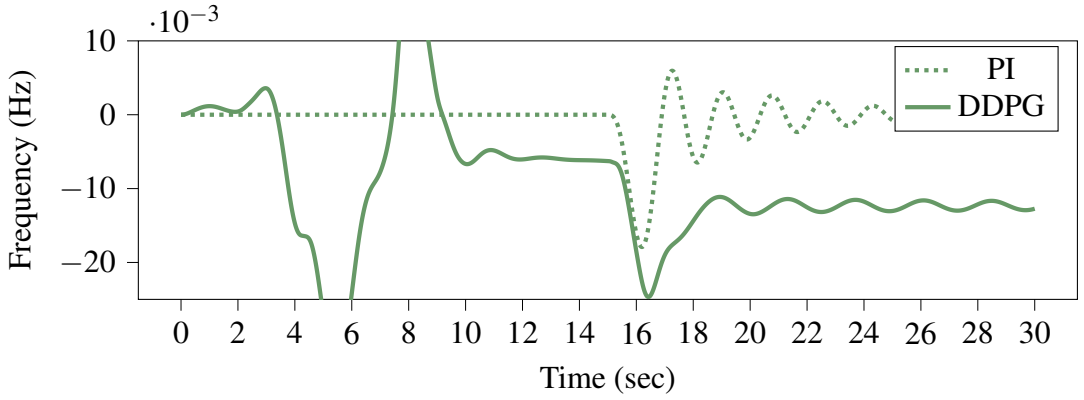

Figure 6.30: text

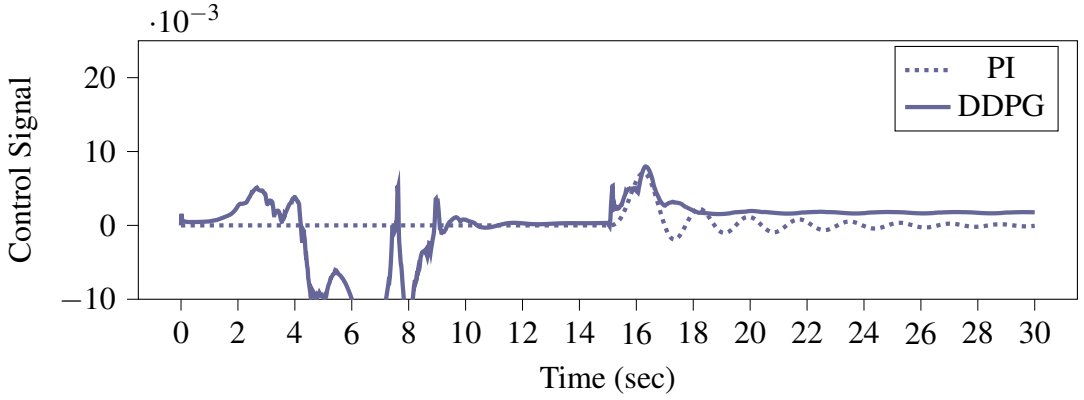

Figure 6.31: text

6.8 Stochastic Demand Load Profile Experiment

Table 6.8: An overview of actor and critic neural network architectures used in experiments undertaken by Lillicrap *et alias*

Layer	Actor		Critic	
	Activation Function	**Number of Perceptrons**	**Activation Function**	**Number of Perceptrons**
Input Layer	None	3	None	3
Hidden Layer 1	ReLU	400	ReLU	400
Hidden Layer 2	ReLU	300	ReLU	300
Output Layer	tanh	2	None	2

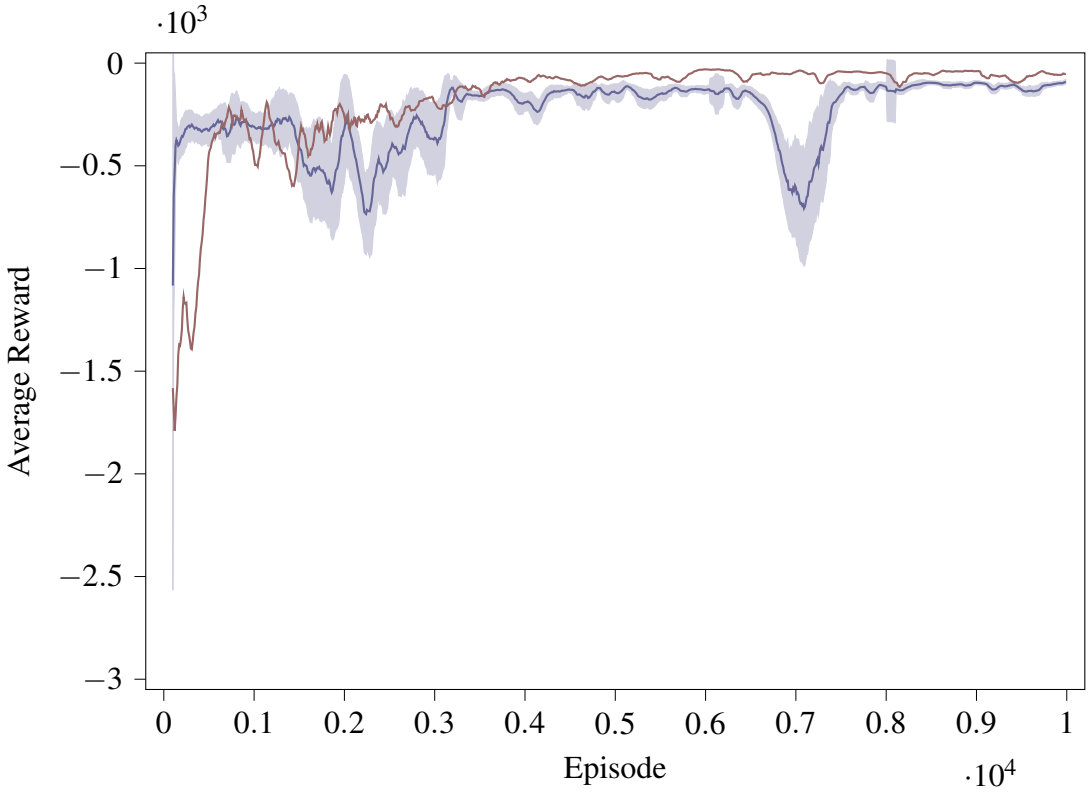

Figure 6.32: text

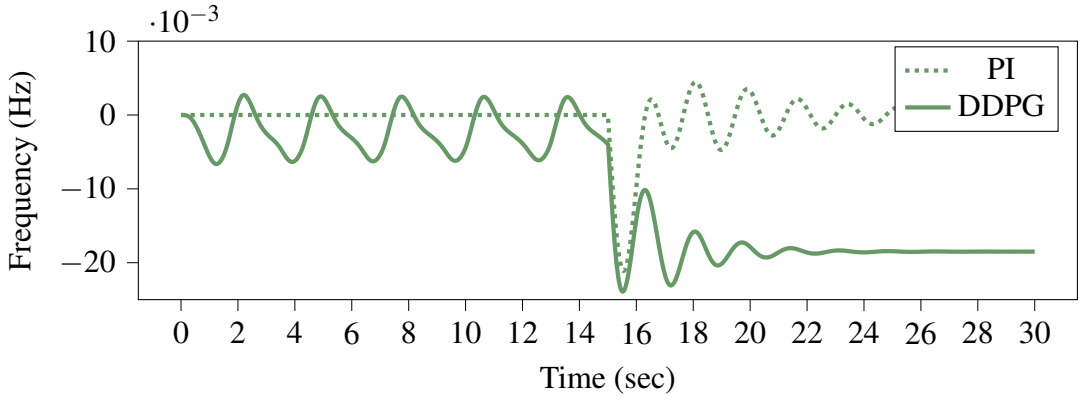

Figure 6.33: text

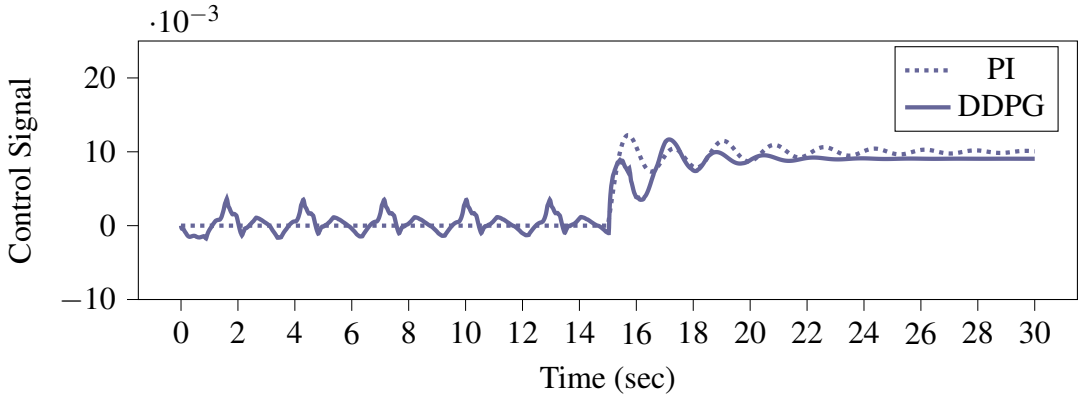

Figure 6.34: text

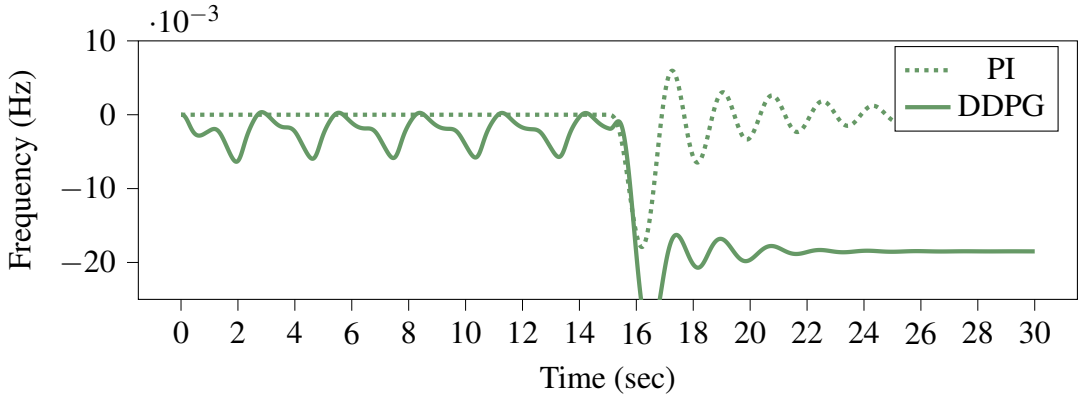

Figure 6.35: text

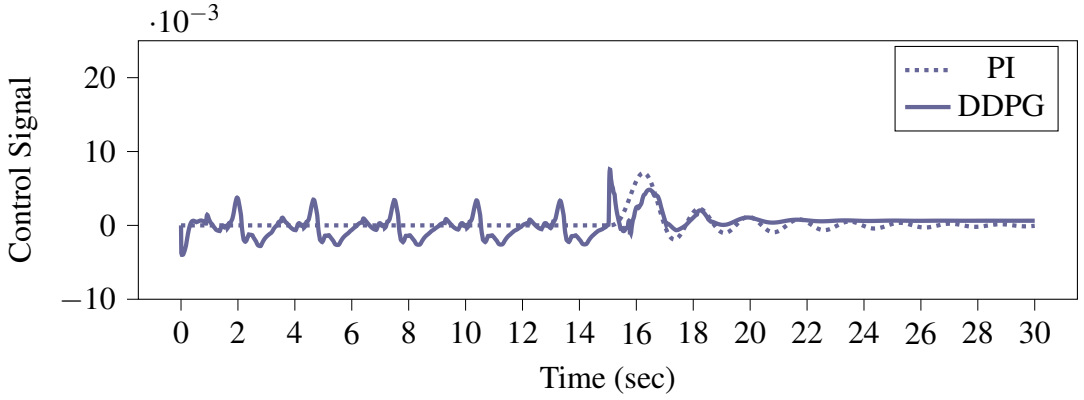

Figure 6.36: text

Chapter 7

Discussion and Future Directions

Monotonically increasing curves in subplot (a) and (b) are desirable, and indicate stable learning for a given reward function, and set of hyperparameters

Reward functions are designed to achieve desirable control behaviour from the agent. The task of controlling frequency and power interchange for a two area power system should therefore have a reward structure that will incentivise the agent to minimise frequency deviations and minimise deviations in the tie-line power interchange from scheduled values. Moreover, the agent should be discouraged from letting the system see large frequency excursions that would activate system protection settings in a real world power system. Finally, it would be useful if the agent develops policies that meet the objectives outlined above, without requiring acute control signals.

Whilst there are no restrictions on reward function specification, it is widely acknowledged that poor reward function selection can impede agent learning, or produce undesirable behaviour. Evidence of unstable and divergent learning can be observed in experiment A and B, as shown in subplot (a) and (b), in Figure **??** and **??**, respectively. This resulted in the DDPG agent developing poor control policies that saturated control actions, as shown in subplot (e) and (f). Agent frequency control performance saw unacceptably large deviations from scheduled values, as shown in subplot (c) and (d).

Experiments C, D, and E removed early termination conditions and associated penalties, resulting in an improvement to agent learning stability, as shown in subplot (a) and (b), in Figure **??**, **??**, and **??**, respectively. Control policies developed under these conditions no longer saturated control actions, as shown in subplot (e) and (f). This led to an improvement in agent control performance, comparable to the optimally tuned PI controller, shown in subplot (c) and (d).

After reviewing agent performance during training it was observed that control signals in experiments A to E were noisy. OU noise is added to control signals in order to help the DDPG agent explore the policy and state-action value spaces to reduce the probability of convergence on sub-optimal policies; however, excessive noise settings have been shown to impede agent learning during the later stages of training. Reducing the OU noise

parameters by a factor of 10 in experiment F saw agent frequency control performance closely resemble the frequency control performance of the optimally tuned PI controller, shown in subplot (c) and (d), for Figure **??**. This result shows the feasibility for using DDPG trained neural networks in controlling the frequency of a two area power system.

This result needs to be tested further by making additional reward function modifications, and minor changes to noise processes. Further, investigation of these two aspects should be undertaken using a more rigorous and scientific approach. Research into experimental approaches for reward function, and hyperparameter testing would ensure that research is conducted in a systematic providing better insight to the problem. Research should also be undertaken to determine a better approach to reporting agent learning, and measuring control performance of the trained agent.

Once settings for which DDPG learns and performs optimally are understood, experiments involving stochastic load demand and non-linear plant models should commence. Contact has been made with Territory Generation and Power and Water Corporation, with the understanding that stochastic load demand profile can be provided; however, the issues such as conditions under which the data can be used and the data sensitivity have yet to be discussed.

Chapter 8

Conclusion

Chapter 9

Future Work

Bibliography

[1] (2020). Electricity generation, Department of industry science energy and resources, [Online]. Available: https://www.energy.gov.au/data/electricity-generation.

[2] "Special report on renewable energy sources and climate change mitigation," Intergovernmental Panel on Climate Change, Tech. Rep., 2012. [Online]. Available: https://www.ipcc.ch/site/assets/uploads/2018/03/SRREN_FD_SPM_final-1.pdf.

[3] "Independent investigation of alice springs system black incident on 13 october 2019," Utilities commission of the northen territory, Tech. Rep., 2019. [Online]. Available: https://utilicom.nt.gov.au/__data/assets/pdf_file/0011/767783/Independent-Investigation-of-Alice-Springs-System-Black-Incident-on-13-October-2019-Report.pdf.

[4] D. Wilkey, "Alice springs system black 13 october 2019," Entura, Tech. Rep., 2019. [Online]. Available: https://utilicom.nt.gov.au/__data/assets/pdf_file/0012/767784/Advice-Entura-Alice-Springs-System-Black-13-October-2019-Report.pdf.

[5] M. Glavic, "Deep reinforcement learning for electric power system control and related problems: A short review and perspectives," *Annual reviews in control*, 2019.

[6] A. J. Wood, B. F. Wollenberg, and G. B. Shelbe, *Power generation, operation, and control*, 3rd Edition. Wiley, 2013.

[7] "Power system frequency and time deviation monitoring report - reference guide," Australian Energy Market Operator, Tech. Rep., Jul. 2012. [Online]. Available: https://aemo.com.au/-/media/files/electricity/nem/security_and_reliability/ancillary_services/frequency-and-time-error-reports/frequency_report_reference_guide_v2_0.pdf.

[8] *Network technical code and network planning criteria*, Power and Water Corporation, 2013. [Online]. Available: https://www.powerwater.com.au/__data/assets/pdf_file/0022/5962/Power-and-Water-Corporation-Network-Technical-Code-and-Network-Planning-Criteria.pdf.

[9] P. C. Sen, *Principles of Electric Machines and Power Electronics*, 3rd Edition. Wiley, 2014.

[10] "Power system frequency risk review - final report," Australian Energy Market Operator, Tech. Rep., Apr. 2018. [Online]. Available: https://aemo.com.au/-/media/files/electricity/nem/planning_and_forecasting/psfrr/2018_power_system_frequency_risk_review-final_report.pdf?la=en&hash=1684259023A1FA274D7F3B8CE855D0BA.

[11] "South australian electricity report," Australian Energy Market Operator, Tech. Rep., Nov. 2019. [Online]. Available: https://www.aemo.com.au/-/media/Files/Electricity/NEM/Planning_and_Forecasting/SA_Advisory/2019/2019-South-Australian-Electricity-Report.pdf.

[12] J. D. Glover, S. S. Mulukutla, and T. J. Overbye, *Power system analysis and design*, 5th Edition. Cengage Learning, 2012.

[13] D. P. Kothari and I. J. Nagrath, *Modern Power System Analysis*, 4th Edition. McGraw Hill India, 2011.

[14] J. J. Grainger and W. D. Stevenson, *Power System Analysis*. McGraw Hill, 1994.

[15] (2020). Ancilliary services, Australian Energy Market Operator, [Online]. Available: https://aemo.com.au/en/energy-systems/electricity/wholesale-electricity-market-wem/system-operations/ancillary-services.

[16] H. Bevrani and T. Hiyama, *Intelligent Automatic Generation Control*. CRC Press, 2011.

[17] P. Kundur, *Power System Stability and Control*. McGraw-Hill Inc., 1994.

[18] R. S. Sutton and A. G. Barto, *Reinforcement Learning*, M. Press, Ed. 2018.

[19] R. Bellman, "A markovian decision process," *Journal of Mathematics and Mechanics*, vol. 6, no. 5, pp. 679–684, 1957.

[20] ——, "The theory of dynamic programming," *Bulletin of the American Mathematical Society*, vol. 60, no. 6, pp. 503–515, 1954. [Online]. Available: https://projecteuclid.org/download/pdf_1/euclid.bams/1183519147.

[21] R. A. Howard, *Dynamic Programming and Markov Processes*. Cambridge, MA: MIT Press, 1960.

[22] R. Bellman and S. E. Dreyfus, "Functional approximations and dynamic programming," *Mathematical Tables and Other Aids to Computation*, 1959.

[23] I. H. Witten, "An adaptive optimal controller for discrete-time markov environments," *Information and Control*, vol. 34, no. 4, pp. 286–295, 1977, ISSN: 0019-9958. DOI: https://doi.org/10.1016/S0019-9958(77)90354-0. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0019995877903540.

[24] P. J. Werbos, "Building and understanding adaptive systems: A statistical/numerical approach to factory automation and brain research," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 17, no. 1, pp. 7–20, 1987.

[25] C. Watkins, "Learning from delayed rewards," PhD thesis, King's College, Cambridge, UK, May 1989. [Online]. Available: http://www.cs.rhul.ac.uk/~chrisw/new_thesis.pdf.

[26] D. Silver, *Lecture notes on reinforcement learning*, 2015.

[27] G. Hinton, L. Deng, D. Yu, G. E. Dahl, A. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. N. Sainath, and B. Kingsbury, "Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups," *IEEE Signal Processing Magazine*, vol. 29, no. 6, pp. 82–97, 2012.

[28] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems 25*, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, Eds., Curran Associates, Inc., 2012, pp. 1097–1105. [Online]. Available: http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf.

[29] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv 1409.1556*, Sep. 2014.

[30] C. Szegedy, Wei Liu, Yangqing Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2015, pp. 1–9.

[31] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016, pp. 770–778.

[32] 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, Feb. 2015, ISSN: 1476-4687. DOI: 10.1038/nature14236. [Online]. Available: https://doi.org/10.1038/nature14236.

[33] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, *Continuous control with deep reinforcement learning*, 2015. arXiv: 1509.02971 [cs.LG].

[34] J. Schulman, S. Levine, P. Abbeel, M. Jordan, and P. Moritz, "Trust region policy optimization," in *Proceedings of the 32nd International Conference on Machine Learning*, F. Bach and D. Blei, Eds., ser. Proceedings of Machine Learning Research, vol. 37, Lille, France: PMLR, Jul. 2015, pp. 1889–1897. [Online]. Available: http://proceedings.mlr.press/v37/schulman15.html.

[35] J. Schulman, P. Moritz, S. Levine, M. Jordan, and P. Abbeel, *High-dimensional continuous control using generalized advantage estimation*, 2015. arXiv: 1506.02438 [cs.LG].

[36] S. Haykin, *Neural Networks: A Comprehensive Foundation*. Prentice Hall, 1999.

[37] F. Rosenblatt, "The perceptron - a perceiving and recognizing automaton," Cornell Aeronautical Laboratory, 1957.

[38] ——, "The perceptron: A probabilistic model for information storage and organization in the brain.," *Psychological Review*, vol. 65, no. 6, pp. 386–408, 1958. DOI: 10.1037/h0042519. [Online]. Available: https://doi.org/10.1037/h0042519.

[39] M. Minsky and S. Papert, *Perceptrons*, ser. Perceptrons. Oxford, England: M.I.T. Press, 1969.

[40] P. J. Werbos, "Applications of advances in nonlinear sensitivity analysis," in *System Modeling and Optimization*, R. F. Drenick and F. Kozin, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 1982, pp. 762–770, ISBN: 978-3-540-39459-4.

[41] S. Hochreiter, "Untersuchungen zu dynamischen neuronalen netzen," *Diploma, Technische Universität München*, vol. 91, no. 1, 1991.

[42] J. N. Tsitsiklis and B. Van Roy, "An analysis of temporal-difference learning with function approximation," *IEEE Transactions on Automatic Control*, vol. 42, no. 5, pp. 674–690, 1997.

[43] D. Silver, G. Lever, N. Heess, T. Degris, D. Wierstra, and M. Riedmiller, "Deterministic policy gradient algorithms," in *Proceedings of the 31st International Conference on Machine Learning*, E. P. Xing and T. Jebara, Eds., ser. Proceedings of Machine Learning Research, vol. 32, Bejing, China: PMLR, Jun. 2014, pp. 387–395. [Online]. Available: http://proceedings.mlr.press/v32/silver14.html.

[44] I. Prabhat and D. P. Kothari, "Recent philosophies of automatic generation control strategies in power systems," *IEEE Transactions on Power Systems*, vol. 20, no. 1, Feb. 2005.

[45] N. Cohn, "The evolution of real time control applications to power systems," *IFAC Proceedings Volumes*, vol. 16, no. 1, pp. 1–17, 1983.

[46] H. Saadat, *Power System Analysis*, 3rd. PSA Publishing LLC, 2011.

[47] C. Concordia and L. K. Kirchmayer, "Tie-line power and frequency control of electric power systems," *Transactions of the American Institute of Electrical Engineers. Part III: Power Apparatus and Systems*, vol. 72, no. 3, pp. 562–572, 1953.

[48] N. Cohn, "Techniques for improving the control of bulk power transfers on interconnected systems," *IEEE Transactions on Power Apparatus and Systems*, vol. PAS-90, pp. 2409–2419, 6 Nov. 1971, ISSN: 0018-9510. DOI: 10.1109/TPAS.1971.292851.

[49] R. P. Aggarwal and F. R. Bergseth, "Large signal dynamics of load-frequency control systems and their optimization using nonlinear programming: I," *IEEE Transactions on Power Apparatus and Systems*, vol. PAS-87, pp. 527–532, 2 Feb. 1968, ISSN: 0018-9510. DOI: 10.1109/TPAS.1968.292049.

[50] ——, "Large signal dynamics of load-frequency control systems and their optimization using nonlinear programming: Ii," *IEEE Transactions on Power Apparatus and Systems*, vol. PAS-87, pp. 532–538, 2 Feb. 1968, ISSN: 0018-9510. DOI: 10.1109/TPAS.1968.292050.

[51] N. Cohn, "Some aspects of tie-line bias control on interconnected power systems," *Transactions of the American Institute of Electrical Engineers. Part III: Power Apparatus and Systems*, vol. 75, pp. 1415–1436, 3 Jan. 1956, ISSN: 2379-6766. DOI: 10.1109/AIEEPAS.1956.4499454.

[52] O. I. Elgerd and C. E. Fosha, "Optimum megawatt-frequency control of multiarea electric energy systems," *IEEE Transactions on Power Apparatus and Systems*, no. 4, pp. 556–563, 1970.

[53] K. Ogata, *Modern Control Engineering*, 5th ed. Pearson, 2010.

[54] T. E. Bechert and N. Chen, "Area automatic generation control by multi-pass dynamic programming," *IEEE Transactions on Power Apparatus and Systems*, vol. 96, pp. 1460–1469, 5 Sep. 1977, ISSN: 0018-9510. DOI: 10.1109/T-PAS.1977.32474.

[55] C. Concordia, L. K. Kirchmayer, and E. A. Szymanski, "Effect of speed-governor dead band on tie-line power and frequency control performance," *Transactions of the American Institute of Electrical Engineers. Part III: Power Apparatus and Systems*, vol. 76, no. 3, pp. 429–434, Apr. 1957, ISSN: 2379-6766. DOI: 10.1109/AIEEPAS.1957.4499581.

[56] H. G. Kwatny, K. C. Kalnitsky, and A. Bhatt, "An optimal tracking approach to load-frequency control," *IEEE Transactions on Power Apparatus and Systems*, vol. 94, no. 5, pp. 1635–1643, Sep. 1975, ISSN: 0018-9510. DOI: 10.1109/TPAS.1975.32006. [Online]. Available: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1601608.

[57] O. Elgerd, *Electric energy systems theory: an introduction*, 2nd ed. McGraw-Hill, 1994.

[58] J. Morsali, K. Zare, and M. Tarafdar Hagh, "Appropriate generation rate constraint (grc) modeling method for reheat thermal units to obtain optimal load frequency controller (lfc)," in *2014 5th Conference on Thermal Power Plants (CTPP)*, Jun. 2014, pp. 29–34. DOI: 10.1109/CTPP.2014.7040611.

[59] C. S. Chang and W. Fu, "Area load frequency control using fuzzy gain scheduling of pi controllers," *Electric Power Systems Research*, vol. 42, no. 2, pp. 145–152, Aug. 1997. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0378779696011996.

[60] E. Cam and I. Kocaarslan, "Load frequency control in two area power system using fuzzy logic controller," *Energy Conversion and Management*, vol. 46, no. 2, pp. 233–243, Jan. 2005. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0196890404000779.

[61] E. Yesil, M. Guzelkaya, and I. Eksin, "Self tuning pid type load and frequency controller," *Energy Conversion*, vol. 45, no. 3, pp. 377–390, Feb. 2004. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0196890403001493.

[62] P. J. Fleming and C. M. Fonseca, "Genetic algorithms in control systems engineering," *IFAC Proceedings Volumes*, vol. 26, no. 2, Part 2, pp. 605–612, 1993, 12th Triennal Wold Congress of the International Federation of Automatic control. Volume 2 Robust Control, Design and Software, Sydney, Australia, 18-23 July, ISSN: 1474-6670. DOI: https://doi.org/10.1016/S1474-6670(17)49015-X. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S147466701749015X.

[63] C. S. Chang, W. Fu, and F. Wen, "Load frequency control using genetic algorithm based fuzzy gain scheduling of pi controllers," *Electric Machines and Power Systems*, vol. 26, no. 1, 1998. [Online]. Available: https://www.tandfonline.com/doi/abs/10.1080/07313569808955806.

[64] D. Rerkpreedapong, A. Hasanovic, and A. Feliachi, "Robust load frequency control using genetic algorithms and linear matrix inequalities," *IEEE Transactions on Power Systems*, vol. 18, pp. 855–861, 2 May 2003, ISSN: 1558-0679. DOI: 10.1109/TPWRS.2003.811005.

[65] S. P. Ghoshal, "Application of ga/ga-sa based fuzzy automatic generation control of a multi-area thermal generating system," *Electric Power Systems Research*, vol. 70, no. 2, pp. 115–127, 2004, ISSN: 0378-7796. DOI: https://doi.org/10.1016/j.epsr.2003.11.013. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0378779603002980.

[66] F. Beaufays, Y. Abdel-Magid, and B. Widrow, "Application of neural networks to load-frequency control in power systems," *Neural Networks*, vol. 7, no. 1, pp. 183–194, 1994, ISSN: 0893-6080. DOI: https://doi.org/10.1016/0893-6080(94)90067-1. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0893608094900671.

[67] A. Demiroren, N. S. Sengor, and H. L. Zeynelgil, "Automatic generation control by using ann technique," *Electric Power Components and Systems*, vol. 29, no. 10, pp. 883–896, 2001. DOI: 10.1080/15325000152646505. eprint: https://doi.org/10.1080/15325000152646505. [Online]. Available: https://doi.org/10.1080/15325000152646505.

[68] H. L. Zeynelgil, A. Demiroren, and N. S. Sengor, "The application of ann technique to automatic generation control for multi-area power system," *International Journal of Electrical Power and Energy Systems*, vol. 24, no. 5, pp. 345–354, 2002, ISSN: 0142-0615. DOI: https://doi.org/10.1016/S0142-0615(01)00049-7. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0142061501000497.

[69] Z. Zhang, D. Zhang, and R. C. Qiu, "Deep reinforcement learning for power system: An overview," *CSEE Journal of Power and Energy Systems*, 2019.

[70] Z. Yan and Y. Xu, "Data-driven load frequency control for stochastic power systems: A deep reinforcement learning method with continuous action search," *IEEE Transactions on Power Systems*, vol. 34, no. 2, pp. 1653–1656, Mar. 2019, ISSN: 1558-0679. DOI: 10.1109/TPWRS.2018.2881359.

[71] P. Henderson, R. Islam, P. Bachman, J. Pineau, D. Precup, and D. Meger, *Deep reinforcement learning that matters*, 2017. arXiv: 1709.06560 [cs.LG].

[72] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, *Openai gym*, 2016. eprint: `arXiv:1606.01540`.

Appendix A

Frequency Domain Modelling for a Single Area Power System

When considering a single area power system, there are three main components that are often focused on:

1. **Governor:** used for controlling the angular velocity (and frequency) of the system.
2. **Turbine:** this is the steam turbine which provides the mechanical torque to drive the generator.
3. **Generator load:** describes the electrical power that is produced and the electrical torque from connected loads.

A.1 Governor Model

The most important part of a speed governor are the two large masses that rotate on a central axis. These masses are mechanically coupled to the turbine drive shaft, so their angular velocity is a function of the turbine speed. Elgerd and Fosha's text [52] provides a schematic representation of the governing system for a steam turbine, shown in Figure A.1. Borrowing heavily from Kothari [13], this schematic is used to derive the plant model for the governor.

Perturbing point A downward some incremental distance, Δy_A, the turbine power output will change by a directly proportional amount. If ΔP_C is the power increase, this can be expressed as:

$$\Delta y_A = k_C \Delta P_C. \tag{A.1}$$

An increase in ΔP_C will cause the pilot valve to move up, and oil will flow onto the top of the main piston forcing it downwards. As the steam valve opens, more steam will drive the turbine faster causing the governor to lower point B. Mathematically, the movement of C can be expressed as the result of two separate inputs:

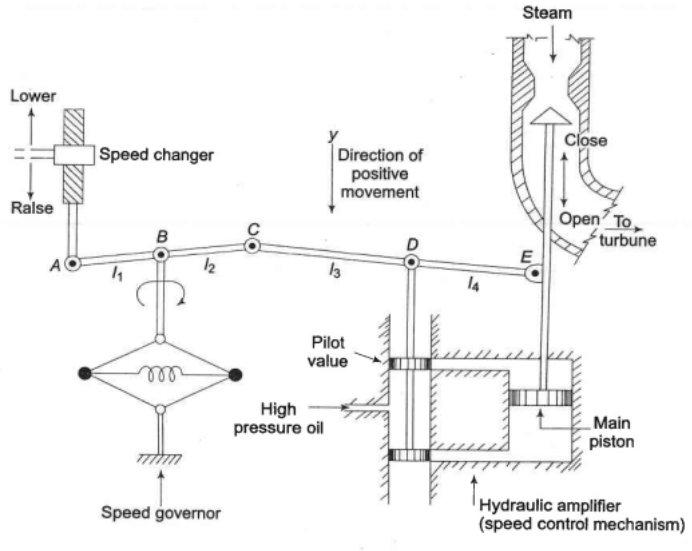

Figure A.1: A schematic of a steam governor

1. Assuming that Δy_A is small, using similar triangles, it can be written that:

$$\Delta y_C = -\frac{l_2}{l_1}\Delta y_A. \tag{A.2}$$

2. For some frequency increase Δf, point B will move downward. Assuming A is fixed, by similar triangles it is clear that:

$$\Delta y_C = \frac{l_1 + l_2}{l_1}\Delta y_B. \tag{A.3}$$

Letting $k_1 = \frac{l_2}{l_1}$, $k_2 = \frac{l_1+l_2}{l_1}k_2'$, and using Equation A.1, the total movement in point C can be expressed as:

$$\Delta y_C = -k_1 k_C \Delta P_C + k_2 \Delta f. \tag{A.4}$$

A similar analysis considering perturbation of point C and E can be undertaken to express the movements of point D. The analysis makes use of similar triangles and results in the expression:

$$\Delta y_D = \frac{l_4}{l3 + l_4}\Delta y_C + \frac{l3}{l_3 + l_4}\Delta y_E. \tag{A.5}$$

Letting $k_3 = \frac{l_4}{l_3+l_4}$ and $k_4 = \frac{l3}{l_3+l_4}$, this can be re-expressed as:

$$\Delta y_D = k_3 \Delta y_C + k_4 \Delta y_E. \tag{A.6}$$

Movement, Δy_D, of point D results in pilot valve ports opening and oil will flowing onto the cylinder causing movement Δy_E. If point D moves up, oil will force point E down, and conversely if point D moves down, oil will force point E upwards. To simplify

the dynamics of this scenario, the following assumptions are made:

1. Inertial reaction forces of the main piston and steam valve are negligible compared to the forces exerted on the piston due to the oil

2. Due to the first assumption, the rate of oil admitted to to the cylinder is proportional to the port opening Δy_D.

Given the assumptions above, the volume of oil admitted to the cylinder is therefore proportional to the time integral of Δy_D. Dividing the oil volume by the cross-sectional area of the piston:

$$\Delta y_E = k_5 \int (-\Delta y_D) dt. \tag{A.7}$$

Taking the Laplace transform of equations A.4, A.6, and A.7 yields:

$$\Delta Y_C(s) = -k_1 k_C \Delta P_C(s) + k_2 \Delta F(s) \tag{A.8}$$

$$\Delta Y_D(s) = k_3 \Delta Y_C(s) + k_4 \Delta Y_E(s) \tag{A.9}$$

$$\Delta Y_E(s) = -k_5 \frac{1}{s} \Delta Y_D(s) \tag{A.10}$$

Algebraically manipulating A.8, A.9, and A.10 eliminates $\Delta Y_C(s)$ and $\Delta Y_D(s)$ and results in the following equation:

$$\Delta Y_E(s) = \frac{k_1 k_3 k_C \Delta P_C(s) - k_2 k_3 \Delta F(s)}{k_4 + \frac{s}{k_5}}. \tag{A.11}$$

Equation A.11 can be re-expressed as:

$$\Delta Y_E(s) = \left[\Delta P_C(s) - \frac{1}{R} \Delta F(s) \right] \times \left(\frac{K_{sg}}{1 + T_{sg} s} \right), \tag{A.12}$$

where

$$R = \frac{k_1 k_C}{k_2} \tag{A.13}$$

$$K_{sg} = \frac{k_1 k_3 k_C}{k_4} \tag{A.14}$$

$$T_{sg} = \frac{1}{T_{sg}}. \tag{A.15}$$

Equation A.12 is a model of the governor in the frequency domain. The parameter R is referred to as the speed regulation of the governor; the parameter K_{sg} is referred to as the gain of the speed governor; and the parameter T_{sg} is referred to as the time constant of the speed governor.

The complete block diagram of the steam governor model can be seen in Figure A.2.

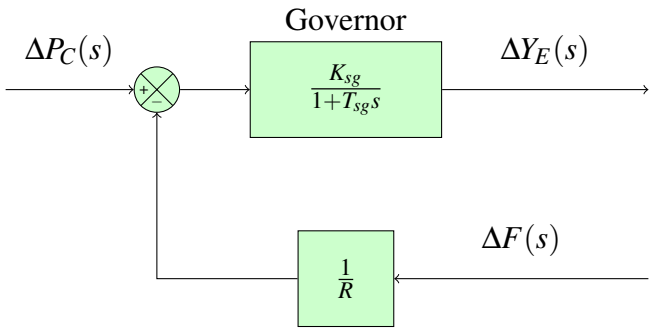

Figure A.2: Block diagram of the steam governor model in the frequency domain

A.2 Turbine Model

To develop a turbine model, consider a fossil-fuelled single reheat tandem-compound turbine. A basic configuration of components can be seen in Figure A.3. The derivation and analysis that follows borrows heavily from Kundar [17]. Steam enters a high pressure (HP) section through the control valve and the inlet piping. Housing for the control valves is called the steam chest. The HP exhaust steam is passed through the re-heater. Reheat steam flows into the IP turbine section through the reheat intercept valve (IV) and the inlet piping. Crossover piping provides a path for the steam from IP section exhaust to the low pressure (LP) inlet.

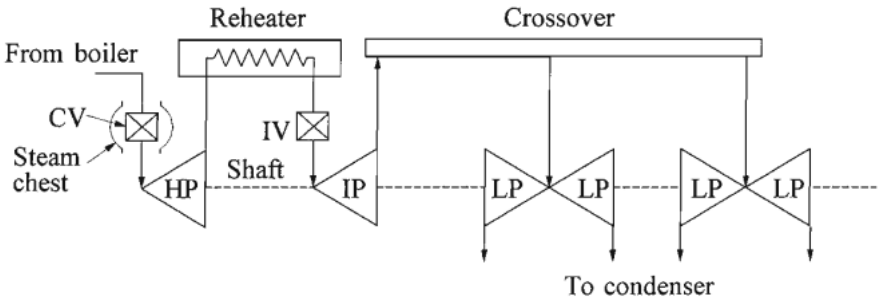

Figure A.3: Configuration of a fossil-fuelled single reheat tandem-compound turbine [17].

Control valve position, y_E, modulates the steam flow through the turbine for load/frequency control during normal operation. The response of steam flow to a change in control valve position exhibits a time constant T_{CH} due to the charging time of the steam chest and the inlet piping to the HP section. An intercept valve is normally used for rapid control of the turbine in the event of an overspeed; however, will not be considered in this analysis. The reheater holds a substantial amount of steam making the dynamics of this component slow enough to require its own time constant T_{RH}. Steam flowing into the LP sections experiences additional time constant T_{CO} associated with the crossover piping. Figure 2 shows the block diagram representation of the tandem compound reheat turbine

[17].

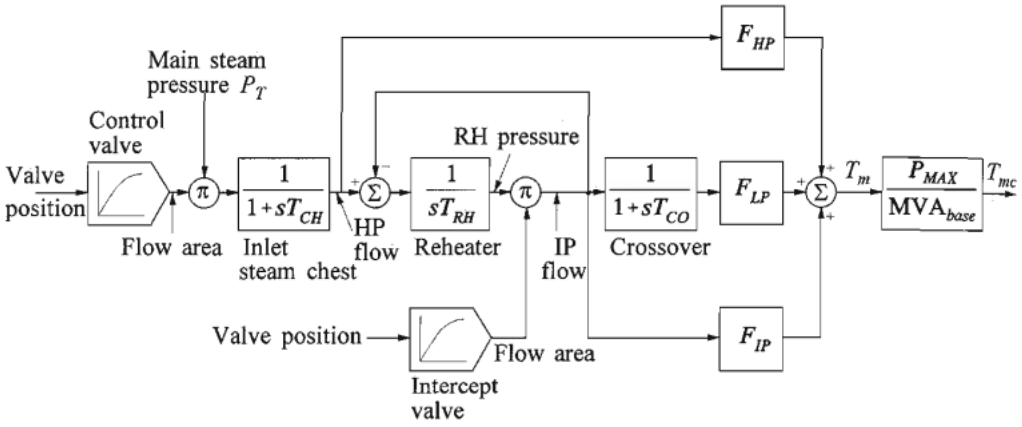

Figure A.4: Block diagram of the tandem-compound reheat turbine [17].

To simplify the model shown in Figure A.4, it is assumed that T_{CO} is negligible in comparison to T_{RH}. Moreover, the remaining two time constants, T_{CH} and T_{RH}, have been combined into a single time constant T_t. Hence, the power output of the turbine, $\Delta P_G(s)$, can be linked to the valve position, $\Delta Y_E(s)$, in the frequency domain with the following expression:

$$\Delta F(s) = \left(\frac{K_t}{1+T_t s}\right) \times P_G(s) \tag{A.16}$$

Equation A.16 is a simplified model of the turbine in the frequency domain. The parameter K_t is referred to as the gain of the turbine; and the parameter T_t is referred to as the time constant of the turbine. The block diagram showing this representation can be seen in Figure A.5.

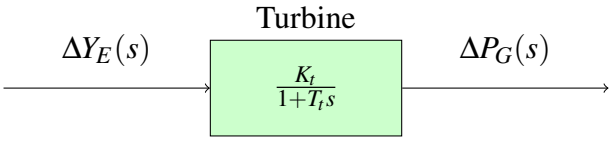

Figure A.5: Simplified block diagram of the tandem compound reheat turbine

A.3 Generator Load Model

The generator-load model derivation presented below borrows heavily from Kothari [13]. If the turbine output is in some steady state, and the power system experiences some perturbation, then the incremental power from the turbine is given by ΔP_G. Noting that the main perturbation experienced by a power system is the change in load demand, ΔP_L, the incremental input to the generator-load system is given by $\Delta P_G - \Delta P_L$.

The increment in power input to the system is accounted for in two ways:

1. Rate of increase of stored kinetic energy in the generator rotor. At scheduled frequency f_0, the stored energy is:

$$(W_{ke})_0 = H \times P_r, \tag{A.17}$$

where P_r is the kilowatt rating of the turbo-generator and H is defined as the inertial constant. Given that the kinetic energy is proportional to the square of the speed (frequency), the kinetic energy at a frequency of $f_0 + \Delta f$ can be written as:

$$W_{ke} = (W_{ke})_0 \left(\frac{f_0 + \Delta f}{f_0} \right)^2 \approx HP_r \left(1 + \frac{2\Delta f}{f_0} \right) \tag{A.18}$$

Therefore, the rate of change of kinetic energy is:

$$\frac{d}{dt}(W_{ke}) = \frac{2HP_r}{f_0} \frac{d}{dt}(\Delta f). \tag{A.19}$$

2. Given that motors make up a reasonable percentage of the load demand it must be considered that the motor load changes as the frequency changes. The rate of change of load with respect to frequency, $\frac{\partial P_L}{\partial f}$, can be considered constant for small changes in frequency Δf and can be expressed as:

$$\frac{\partial P_L}{\partial f} \Delta f = B \Delta f, \tag{A.20}$$

where B can be empirically determined, and is dependent on the proportion of motors that comprise the load demand. Note that where B is positive, then the load is predominantly comprised of motors.

Using Equations A.19 and A.20, the power balance equation for the incremental input to the generator-load system can be written as:

$$\Delta P_G - \Delta P_L = \frac{2HP_r}{f_0} \frac{d}{dt}(\Delta f) + B \Delta f. \tag{A.21}$$

Dividing throughout by P_r yields the following:

$$\Delta P_G(pu) - \Delta P_L(pu) = \frac{2H}{f_0} \frac{d}{dt}(\Delta f) + B(pu) \Delta f. \tag{A.22}$$

Taking the Laplace transform of equation A.22 and rearranging, we get the following expression:

$$\Delta F(s) = \frac{\Delta P_G(s) - \Delta P_L(s)}{B + \frac{2H}{f_0} s}. \tag{A.23}$$

Equation A.23 can re-expressed as:

$$\Delta F(s) = [\Delta P_G(s) - \Delta P_L(s)] \times \left(\frac{K_{gl}}{1 + T_{gl}} \right), \tag{A.24}$$

where

$$K_{gl} = \frac{1}{B} \tag{A.25}$$

$$T_{gl} = \frac{2H}{Bf_0}. \tag{A.26}$$

Equation A.24 describes the generator-load in the frequency domain. The parameter K_{gl} is referred to as the gain of the generator load; and the parameter T_{gl} is referred to as the time constant of the generator load.

The complete block diagram of governor model can be seen in Figure A.6 below.

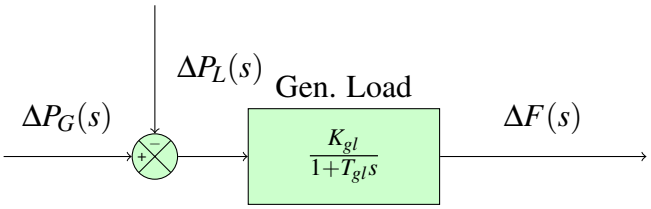

Figure A.6: Block diagram of the generator load model in the frequency domain

Appendix B

Frequency Domain Modelling

B.1 Tie Line Model

The power systems in each of the control areas for a two-area power system are comparable to the single area model shown in Figure 2.10. In fact, the governor, and turbine models are interchangeable between the single area model and the two-area model. We do, however, still have make some changes — these are as follows:

1. model the interaction of the two power systems over the tie line; and
2. re-analyse our generator-load demand model.

The derivations presented below are motivated by Kothari [13].

The transmission infrastructure, or tie line, allows for the flow of power from area 1 to area 2, and vice versa. For the convenience of this derivation, let any symbol with a subscript of 1 refer to power area 1 and those with a subscript 2 refer to power area 2. Now, letting the power angles of the area 1 generator and the area 2 generator be $(\delta_1)_0$ and $(\delta_2)_0$, respectively, we can write an expression for the power transported out of area 1 as:

$$P_{tie,1} = \frac{|V_1||V_2|}{X_{12}} \sin\big((\delta_1)_0 - (\delta_2)_0\big). \tag{B.1}$$

Using incremental changes in δ_1 δ_2, we can determine an incremental change in the tie line power using a Taylor series expansion:

$$\Delta P_{tie,1}(pu) = T_{12}(\Delta\delta_1 - \Delta\delta_2). \tag{B.2}$$

Note that T_{12} is the synchronising coefficient, and is mathematically expressed as:

$$T_{12} = \frac{|V_1||V_2|}{P_{r1}X_{12}} \cos\big((\delta_1)_0 - (\delta_2)_0\big). \tag{B.3}$$

Incremental power angles are integrals of incremental frequencies, owing to the link

between angular velocity and frequency. This allows the re-expression of Equation B.2 as:

$$\Delta P_{tie,1} = 2\pi T_{12}\left(\int \Delta f_1 dt - \int \Delta f_2 dt\right). \tag{B.4}$$

The variables Δf_1 and Δf_2 are incremental frequency changes of areas 1 and 2, respectively.

Power can flow both ways over the tie line — the ideas governing the flow from area 2 to area 1 are symmetrical to those presented above. Hence, we can write:

$$\Delta P_{tie,2} = 2\pi T_{21}\left(\int \Delta f_2 dt - \int \Delta f_1 dt\right). \tag{B.5}$$

Note that the synchronising coefficient, T_{21}, can be expressed in terms of T_{12}, therefore:

$$T_{21} = \frac{|V_2||V_1|}{P_{r2}X_{21}}\cos\left((\delta_2)_0 - (\delta_1)_0\right) = \left(\frac{P_{r1}}{P_{r2}}\right)T_{12} = a_{12}T_{12}. \tag{B.6}$$

Equations B.4 and B.5 describe tie line power flow from area 1 to area 2, and from area 2 to area 1, respectively. To link these expression back to the power system model, first take the inverse Laplace transform of Equation B.4 to provide:

$$\Delta P_{tie,1}(s) = \frac{2\pi T_{12}}{s} \times [\Delta F_1(s) - \Delta F_2(s)]. \tag{B.7}$$

Then take the inverse Laplace transform of Equation B.5, which gives:

$$\Delta P_{tie,2}(s) = -\frac{2\pi a_{12} T_{12}}{s} \times [\Delta F_1(s) - \Delta F_2(s)] \tag{B.8}$$

Equation B.7 and B.8 describe the tieline models for power area 1 and 2, respectively. Note that the input for both B.7 and B.8 is $\Delta F_1(s) - \Delta F_2(s)$, and the transfer functions for each differ by a factor of $-a_{12}$. This symmetry allows for the use of a single tie-line model to represent the interconnection dynamics for a two area power system. The tie line block diagram is shown in Figure B.1.

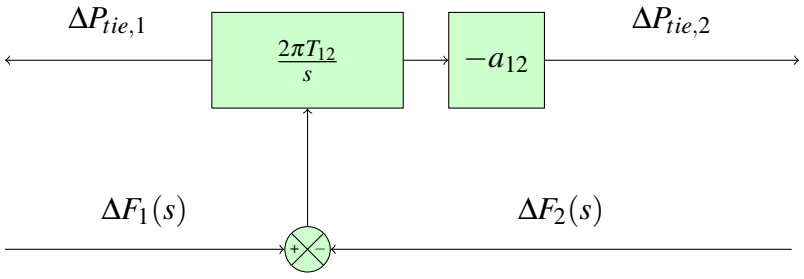

Figure B.1: A block diagram for the tie line connecting power area 1 and power area 2

B.2 Generator Load Model with Tie Line Input

The derivation presented here borrows heavily from Kothari [13]. Analysis in Appendix A.3 reasoned that the incremental power input to the generator load system was given by $\Delta P_G - \Delta P_L$ and that this difference was made up from the rate of increase of stored kinetic energy in the generator rotor, and the additional power drawn by frequency dependent loads such as motors. In the presence of a second power area, this power difference could also come from the power transfer over the tie line. Using this idea, the incremental power balance equation for area 1 can be written as:

$$\Delta P_{G1} - \Delta P_{L1} = \frac{2H_1}{(f_1)_0}\frac{d}{dt}\Delta f_1 + B_1 \Delta f_1 + \Delta P_{tie,1}. \tag{B.9}$$

Additional parameters introduced in Equation B.9 include: the generator inertia, H_1; and the static rate of change in the load demand with respect to frequency due to motors, B_1. Taking the Laplace transform of Equation B.9 and rearranging provides:

$$\Delta F_1(s) = [\Delta P_{G1}(s) - \Delta P_{L1}(s) - \Delta P_{tie,1}(s)] \times \left(\frac{K_{gl1}}{1 + T_{gl1}s}\right), \tag{B.10}$$

where

$$K_{gl1} = \frac{1}{B_1} \tag{B.11}$$

$$T_{gl1} = \frac{2H_1}{B_1(f_1)_0}. \tag{B.12}$$

Equation B.10 is an updated model of the generator-load system which provides for input from the tie line. Note that power could flow either way depending on the scheduled contract for provision of power between area 1 and area 2. Figure B.2 shows the generator-load block diagram with the additional tie line input. Given the symmetry of a two-area power system, an identical update can be applied to the generator load model for area 2.

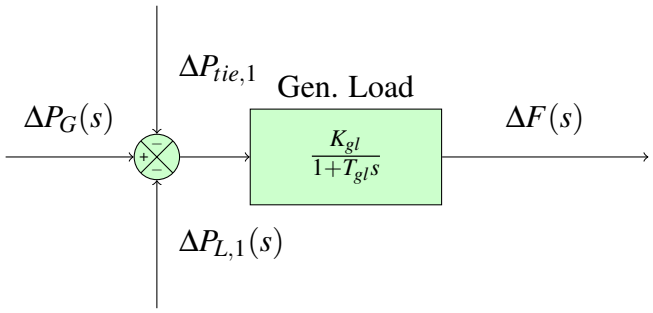

Figure B.2: Generator-load block diagram now has an additional input from the tie line connecting power area 1 and power area 2

The same analysis can be performed for power area 2, which results in the following expression:

$$\Delta F_2(s) = [\Delta P_{G2}(s) - \Delta P_{L2}(s) - \Delta P_{tie,2}(s)] \times \left(\frac{K_{gl2}}{1 + T_{gl2}s} \right), \tag{B.13}$$

where

$$K_{gl2} = \frac{1}{B_2} \tag{B.14}$$

$$T_{gl2} = \frac{2H_2}{B_2(f_2)_0} \tag{B.15}$$

Appendix C

Temporal Domain Modelling

C.1 Temporal Domain Model for a Two Area Power System

To develop a system of first-order, linear, ordinary differential equations for a two area power system, consider only the governor, turbine, generator-load, and tie-line components of Figure 2.14. Let $X_2(s)$, $X_3(s)$, $X_4(s)$, $X_5(s)$, $X_7(s)$, $X_8(s)$, and $X_9(s)$ represent model variables, according to Figure C.1.

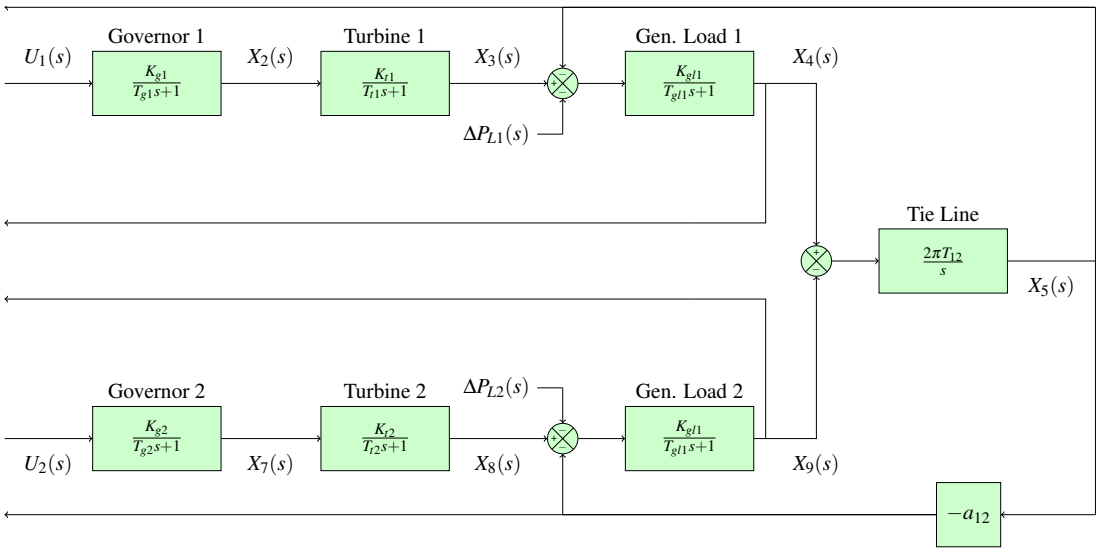

Figure C.1: The governor, turbine, generator-load, and tie-line models for both power areas.

Note that $X_4(s)$ represents the frequency output of area 1; $X_9(s)$ represents the frequency output of area 2; and $X_5(s)$ represents the power flow over the tie-line connecting areas 1 and 2. The system also receives two external control signals that act as inputs the governors in each area. These are denoted as $U_1(s)$ and $U_2(s)$ for control area 1 and control area 2, respectively. Finally, the system experiences changes in load demand as

individuals and industry switch appliances on and off — these are represented by $\Delta P_{L_1}(s)$ and $\Delta P_{L_2}(s)$ for areas 1 and 2, respectively.

Now, using the governor block transfer function in area 1, the following expression can be written:

$$X_2(s) = \frac{K_{sg_1}}{T_{sg_1}s+1}U_1(s). \tag{C.1}$$

Rearranging C.1 and taking the inverse Laplace transform yields the following first order differential equation:

$$\dot{x}_2(t) = \frac{1}{T_{sg_1}}\big(K_{sg_1}u_1(t) - x_2(t)\big). \tag{C.2}$$

Since the two areas of the system are symmetrical, a similar analysis for the same section in area 2 yields:

$$\dot{x}_7(t) = \frac{1}{T_{sg_2}}\big(K_{sg_2}u_2(t) - x_7(t)\big). \tag{C.3}$$

Next, the turbine block transfer function in area 1 can be used to write the following expression:

$$X_3(s) = \frac{K_{t_1}}{T_{t_1}s+1}X_2(s). \tag{C.4}$$

Rearranging and taking the inverse Laplace transform:

$$\dot{x}_3(t) = \frac{1}{T_{t_1}}\big(K_{t_1}x_2(t) - x_3(t)\big). \tag{C.5}$$

Given symmetry in the system a similar equation can be written for area 1:

$$\dot{x}_8(t) = \frac{1}{T_{t_2}}\big(K_{t_2}x_7(t) - x_8(t)\big). \tag{C.6}$$

The same analysis that was undertaken for the governor and turbine can be performed for the generator-load demand block. For area 1, this yields the following first order, ordinary differential equation:

$$\dot{x}_4(t) = \frac{1}{T_{gl_1}}\Big(K_{gl_1}\big(x_3(t) - x_5(t) - \Delta p_{L1}(t)\big) - x_4(t)\Big). \tag{C.7}$$

Since the areas are symmetrical, using C.7 the generator-load model for area two can be written as:

$$\dot{x}_9(t) = \frac{1}{T_{gl_2}}\Big(K_{gl_2}\big(x_8(t) - x_5(t) - \Delta p_{L2}(t)\big) - x_9(t)\Big). \tag{C.8}$$

Finally, the ordinary differential equation for the tie-line block can be expressed as:

$$\dot{x}_5(t) = 2\pi T_{12}\big(x_4(t) - x_9(t)\big). \tag{C.9}$$

C.2 Temporal Domain Model for a PI Controller of a Two Area Power System

The approach taken in section C.1 for deriving the system of differential equations to model the power system, can also be used to model the proportional-integral (PI) feedback controllers. Consider the the proportional and integral feedback loop components of Figure 2.14. Let $X_1(s)$ and $X_6(s)$ represent the output from the integral control block, in area 1 and area 2, as shown in Figures C.2 and C.3, respectively. Note that the controllers receive change in frequency $\Delta F_1(s)$ and $\Delta F_2(s)$ as inputs from the power system model, as well as the change in the tie-line power, $X_5(s)$.

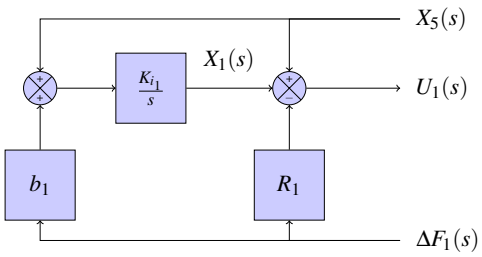

Figure C.2: Proportional integral controller for power area 1, with assigned variables for temporal domain modelling

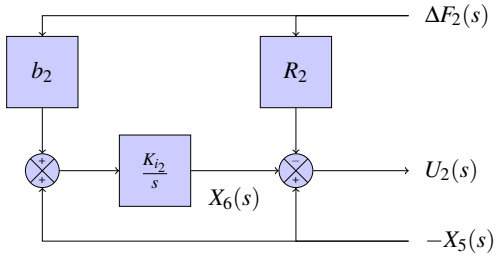

Figure C.3: Proportional integral controller for power area 2, with assigned variables for temporal domain modelling

Considering the integral block for area 1 in Figure C.2, the following expression can be written:

$$X_1(s) = \frac{K_{i_1}}{s} \times \big(b_1 \Delta F_1(s) + X_5(s)\big) \tag{C.10}$$

Rearranging and taking the inverse Laplace transform provides the following differential equation:

$$\dot{x}_1(t) = b_1 \Delta f_1(t) + x_5(t) \tag{C.11}$$

Taking a similar approach for the integral block for area 2 yields:

$$\dot{x}_6(t) = b_2 \Delta f_2(t) - x_5(t) \tag{C.12}$$

Note that equations C.11 and C.12 are not the control outputs that are output to the system, rather, are the differential equations modelling the integral control blocks. The

control output for area 1 and area 2 are shown in equation C.13 and C.14 below.

$$u_1(t) = x_1(t) + x_5(t) - R_1 \Delta f_1(t) \tag{C.13}$$

$$u_2(t) = x_6(t) - x_5(t) - R_2 \Delta f_2(t) \tag{C.14}$$

Appendix D

Implementation of Temporal Models

D.1 Implementation of Power System Model

```
    def int_power_system_sim(self, x_sys, t,
                             control_sig_1, control_sig_2,
                             power_demand_sig_1, power_demand_sig_2,
                             K_sg_1, T_sg_1, K_t_1, T_t_1, K_gl_1, T_gl_1,
                             K_sg_2, T_sg_2, K_t_2, T_t_2, K_gl_2, T_gl_2,
                             T12):
        """
        Inputs
        control sig
        power demand
        power demand
        area one
        area two
        tie line
        """

        # area 1 simulation
        x_2_dot = (1/T_sg_1)*(K_sg_1*control_sig_1 - x_sys[0])
        x_3_dot = (1/T_t_1)*(K_t_1*x_sys[0] - x_sys[1])
        x_4_dot = (K_gl_1/T_gl_1)*(x_sys[1] - x_sys[3] - power_demand_sig_1) - \
                        (1/T_gl_1)*x_sys[2]

        # tie line simulation
        x_5_dot = 2*np.pi*T12*(x_sys[2] - x_sys[6])

        # area 2 simulation
        x_7_dot = (1/T_sg_2)*(K_sg_2*control_sig_2 - x_sys[4])
        x_8_dot = (1/T_t_2)*(K_t_2*x_sys[4] - x_sys[5])
        x_9_dot = (K_gl_2/T_gl_2)*(x_sys[5] + x_sys[3] - power_demand_sig_2) - \
                        (1/T_gl_2)*x_sys[6]

        return x_2_dot, x_3_dot, x_4_dot, x_5_dot, x_7_dot, x_8_dot, x_9_dot
```

D.2 Implementation of PI Contoller Model

```
    def int_control_system_sim(self, x_control_sys, t,
                               frequency_sig_1, frequency_sig_2,
                               tie_line_sig,
                               R_1, K_i_1, b_1,
                               R_2, K_i_2, b_2):
        """
        Inputs
        freq sig
        tie line sig
        area one
        area two
        """

        # controller 1 simulation
        x_1_dot = K_i_1*(tie_line_sig + b_1*frequency_sig_1)

        # controller 2 simulation
        x_6_dot = K_i_2*(-tie_line_sig + b_2*frequency_sig_2)

        return x_1_dot, x_6_dot
```

D.3 Implementation of DDPG Actor Neural Network Model

```
class Actor(nn.Module):
    """Actor (Policy) Model."""

    def __init__(self, state_size, action_size, seed,
                 fc1_units=400, fc2_units=300):
        """Initialize parameters and build model.
        Params
        ======
            state_size (int): Dimension of each state
            action_size (int): Dimension of each action
            seed (int): Random seed
            fc1_units (int): Number of nodes in first hidden layer
            fc2_units (int): Number of nodes in second hidden layer
        """
        super(Actor, self).__init__()
        self.seed = torch.manual_seed(seed)
        self.fc1 = nn.Linear(state_size, fc1_units)
        self.fc2 = nn.Linear(fc1_units, fc2_units)
        self.fc3 = nn.Linear(fc2_units, action_size)
        self.reset_parameters()

    def reset_parameters(self):
        self.fc1.weight.data.uniform_(*hidden_init(self.fc1))
        self.fc2.weight.data.uniform_(*hidden_init(self.fc2))
        self.fc3.weight.data.uniform_(-3e-3, 3e-3)

    def forward(self, state):
        """Build an actor (policy) network that maps states -> actions."""
        x = F.relu(self.fc1(state))
        x = F.relu(self.fc2(x))
```

```
        return torch.tanh(self.fc3(x))
```

D.4 Implementation of DDPG Critic Neural Network Model

```
class Critic(nn.Module):
    """Critic (Value) Model."""

    def __init__(self, state_size, action_size, seed,
                 fcs1_units=400, fc2_units=300):
        """Initialize parameters and build model.
        Params
        ======
            state_size (int): Dimension of each state
            action_size (int): Dimension of each action
            seed (int): Random seed
            fcs1_units (int): Number of nodes in the first hidden layer
            fc2_units (int): Number of nodes in the second hidden layer
        """
        super(Critic, self).__init__()
        self.seed = torch.manual_seed(seed)
        self.fcs1 = nn.Linear(state_size, fcs1_units)
        self.fc2 = nn.Linear(fcs1_units+action_size, fc2_units)
        self.fc3 = nn.Linear(fc2_units, 1)
        self.reset_parameters()

    def reset_parameters(self):
        self.fcs1.weight.data.uniform_(*hidden_init(self.fcs1))
        self.fc2.weight.data.uniform_(*hidden_init(self.fc2))
        self.fc3.weight.data.uniform_(-3e-3, 3e-3)

    def forward(self, state, action):
        """
        Build a critic (value) network that maps
           (state, action) pairs -> Q-values.
        """
        xs = F.relu(self.fcs1(state))
        x = torch.cat((xs, action), dim=1)
        x = F.relu(self.fc2(x))
        return self.fc3(x)
```

D.5 Implementation of Alternative DDPG Actor Neural Network Model

```
class Actor(nn.Module):
    """Actor (Policy) Model."""

    def __init__(self, state_size, action_size, seed, fc_units=256):
        """Initialize parameters and build model.
        Params
        ======
            state_size (int): Dimension of each state
```

```
            action_size (int): Dimension of each action
            seed (int): Random seed
            fc1_units (int): Number of nodes in first hidden layer
            fc2_units (int): Number of nodes in second hidden layer
        """
        super(Actor, self).__init__()
        self.seed = torch.manual_seed(seed)
        self.fc1 = nn.Linear(state_size, fc_units)
        self.fc2 = nn.Linear(fc_units, action_size)
        self.reset_parameters()

    def reset_parameters(self):
        self.fc1.weight.data.uniform_(*hidden_init(self.fc1))
        self.fc2.weight.data.uniform_(-3e-3, 3e-3)

    def forward(self, state):
        """Build an actor (policy) network that maps states -> actions."""
        x = F.relu(self.fc1(state))
        return torch.tanh(self.fc2(x))
```

D.6 Implementation of Alternative DDPG Critic Neural Network Model

```
class Critic(nn.Module):
    """Critic (Value) Model."""

    def __init__(self, state_size, action_size,
                 seed, fcs1_units=256, fc2_units=256, fc3_units=128):
        """Initialize parameters and build model.
        Params
        ======
            state_size (int): Dimension of each state
            action_size (int): Dimension of each action
            seed (int): Random seed
            fcs1_units (int): Number of nodes in the first hidden layer
            fc2_units (int): Number of nodes in the second hidden layer
        """
        super(Critic, self).__init__()
        self.seed = torch.manual_seed(seed)
        self.fcs1 = nn.Linear(state_size, fcs1_units)
        self.fc2 = nn.Linear(fcs1_units+action_size, fc2_units)
        self.fc3 = nn.Linear(fc2_units, fc3_units)
        self.fc4 = nn.Linear(fc3_units, 1)
        self.reset_parameters()

    def reset_parameters(self):
        self.fcs1.weight.data.uniform_(*hidden_init(self.fcs1))
        self.fc2.weight.data.uniform_(*hidden_init(self.fc2))
        self.fc3.weight.data.uniform_(*hidden_init(self.fc3))
        self.fc4.weight.data.uniform_(-3e-3, 3e-3)

    def forward(self, state, action):
        """
        Build a critic (value) network that maps
        (state, action) pairs -> Q-values.
```

```
        """
        xs = F.leaky_relu(self.fcs1(state))
        x = torch.cat((xs, action), dim=1)
        x = F.leaky_relu(self.fc2(x))
        x = F.leaky_relu(self.fc3(x))
        return self.fc4(x)
```

D.7 Implementation of DDPG Controller Class

```
class DdpgController():
    """Interacts with and learns from the environment."""

    def __init__(self, state_size, action_size, random_seed):
        """Initialize an Agent object.

        Params
        ======
            state_size (int): dimension of each state
            action_size (int): dimension of each action
            random_seed (int): random seed
        """
        self.state_size = state_size
        self.action_size = action_size
        self.seed = random.seed(random_seed)

        # Actor Network (w/ Target Network)
        self.actor_local = Actor(state_size, action_size, random_seed).to(device)
        self.actor_target = Actor(state_size, action_size, random_seed).to(device)
        self.actor_optimizer = optim.Adam(self.actor_local.parameters(), lr=LR_ACTOR)

        # Critic Network (w/ Target Network)
        self.critic_local = Critic(state_size, action_size, random_seed).to(device)
        self.critic_target = Critic(state_size, action_size, random_seed).to(device)
        self.critic_optimizer = optim.Adam(self.critic_local.parameters(),
                                           lr=LR_CRITIC, weight_decay=WEIGHT_DECAY)

        # Noise process
        self.noise = OUNoise(action_size, random_seed)

        # Replay memory
        self.memory = ReplayBuffer(action_size, BUFFER_SIZE, BATCH_SIZE, random_seed)

    def step(self, state, action, reward, next_state, done):
        """
        Save experience in replay memory, and use random
        sample from buffer to learn.
        """

        # Save experience / reward
        self.memory.add(state, action, reward, next_state, done)

        # Learn, if enough samples are available in memory
        if len(self.memory) > BATCH_SIZE:
            experiences = self.memory.sample()
            self.learn(experiences, GAMMA)
```

```
    def act(self, state, add_noise=True):
        """Returns actions for given state as per current policy."""
        state = torch.from_numpy(state).float().to(device)
        self.actor_local.eval()
        with torch.no_grad():
            action = self.actor_local(state).cpu().data.numpy()
        self.actor_local.train()
        if add_noise:
            action += self.noise.sample()
        return np.clip(action, -1, 1)

    def reset(self):
        self.noise.reset()

    def learn(self, experiences, gamma):
        """Update policy and value parameters using given batch of
        experience tuples. Q_targets = r + gamma * critic_target(next_state,
        actor_target(next_state))
        where:
            actor_target(state) -> action
            critic_target(state, action) -> Q-value

        Params
        ======
            experiences (Tuple[torch.Tensor]): tuple of (s, a, r, s', done)
            tuples gamma (float): discount factor
        """
        states, actions, rewards, next_states, dones = experiences

        # ---------------------------- update critic ---------------------------- #
        # Get predicted next-state actions and Q values from target models
        actions_next = self.actor_target(next_states)
        Q_targets_next = self.critic_target(next_states, actions_next)
        # Compute Q targets for current states (y_i)
        Q_targets = rewards + (gamma * Q_targets_next * (1 - dones))
        # Compute critic loss
        Q_expected = self.critic_local(states, actions)
        critic_loss = F.mse_loss(Q_expected, Q_targets)
        # Minimize the loss
        self.critic_optimizer.zero_grad()
        critic_loss.backward()
        self.critic_optimizer.step()

        # ---------------------------- update actor ---------------------------- #
        # Compute actor loss
        actions_pred = self.actor_local(states)
        actor_loss = -self.critic_local(states, actions_pred).mean()
        # Minimize the loss
        self.actor_optimizer.zero_grad()
        actor_loss.backward()
        self.actor_optimizer.step()

        # ----------------------- update target networks ----------------------- #
        self.soft_update(self.critic_local, self.critic_target, TAU)
        self.soft_update(self.actor_local, self.actor_target, TAU)

    def soft_update(self, local_model, target_model, tau):
        """Soft update model parameters.
        theta_target = tau*theta_local + (1 - tau)*theta_target
```

```
        Params
        ======
            local_model: PyTorch model (weights will be copied from)
            target_model: PyTorch model (weights will be copied to)
            tau (float): interpolation parameter
        """
        for target_param, local_param in zip(target_model.parameters(),
                                             local_model.parameters()):
            target_param.data.copy_(tau*local_param.data +
                                    (1.0-tau)*target_param.data)
```