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

S. Reynolds (BMa., BFin.)

This interim report submitted as part of the assessment schedule for:

BACHELOR OF ENGINEERING HONOURS (ELECTRICAL)

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

Charles Darwin University
College of Engineering

May 30, 2020

Declaration

I, *S. Reynolds (BMa., BFin.)*, declare that this interim report 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.)

May 30, 2020

Abstract

Power systems are non-linear; however, traditional control architectures for maintaining power system frequency are designed under the assumption that plant can be modelled using linear ordinary differential equations. This assumption is reasonable for small frequency deviations given present levels 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 more 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 the actions it must take to achieve some control objective. It does this with no prior knowledge of the system it is trying to control. This makes RL an ideal tool 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 5 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 know as deep reinforcement learning (DRL), and is considered state-of-the-art for robotic applications.

The objective of this thesis is to apply a DRL algorithm, the Deep Deterministic Policy Gradient (DDPG), to control 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 are representative of aggregated generation and aggregated loads demand for the respective areas. A single transmission line connects the two areas. Load demand for each power area fluctuates as individuals and industry switch appliances on and off. If left unchecked this creates movements 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. It's control action will arrest frequency deviations, and restore each area to the scheduled value, under normal operating conditions.

The performance of the DDPG controller will be evaluated against classical control approaches to provide an assessment of the viability of DRL controllers for this application.

Contents

List of Figures

List of Tables

List of Abbreviations

List of Symbols

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].

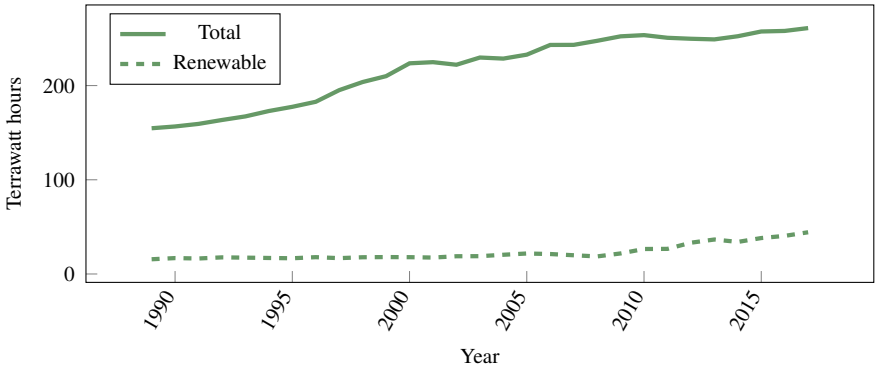

Figure 1.1: Power generation from renewable sources (dashed line), and total power generation (solid line) in Australia from 1977 to 2018.

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, that 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 controller 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 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 details the 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).

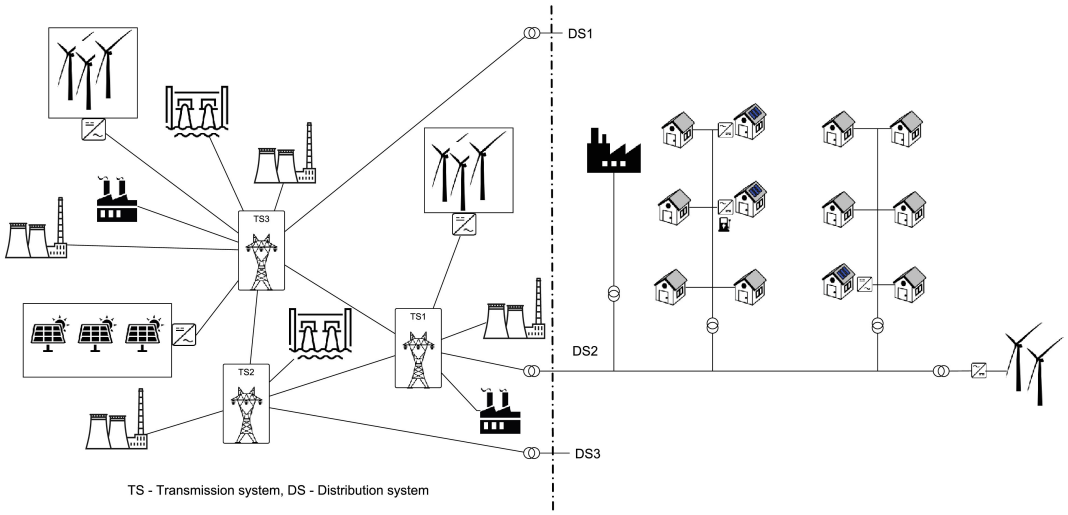

Figure 2.1: A single line diagram of a typical power system. The image shows points of generation from thermal and renewable sources, and the subsequent supply of generated energy to meet load demand through the transmission and distribution network [5].

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.

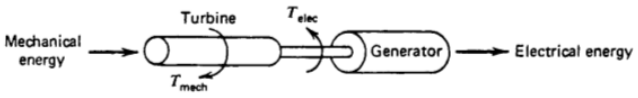

Figure 2.2: A thermal generation unit consisting of a prime mover (turbine), and a synchronous machine [6].

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 businesses and households 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].

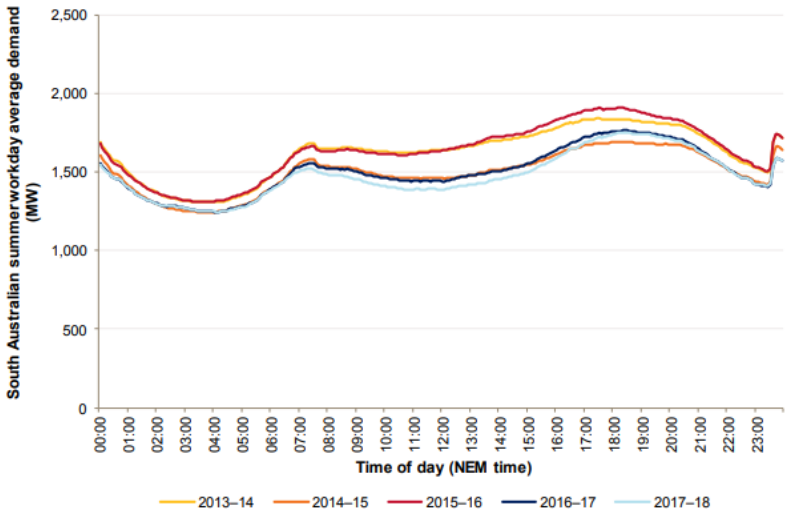

Figure 2.3: Weekday energy demand profile in South Australia during summer [11].

Protection schemes tripping equipment from the network is undesirable as this can leave households and industry 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 AGC is the name used for 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].

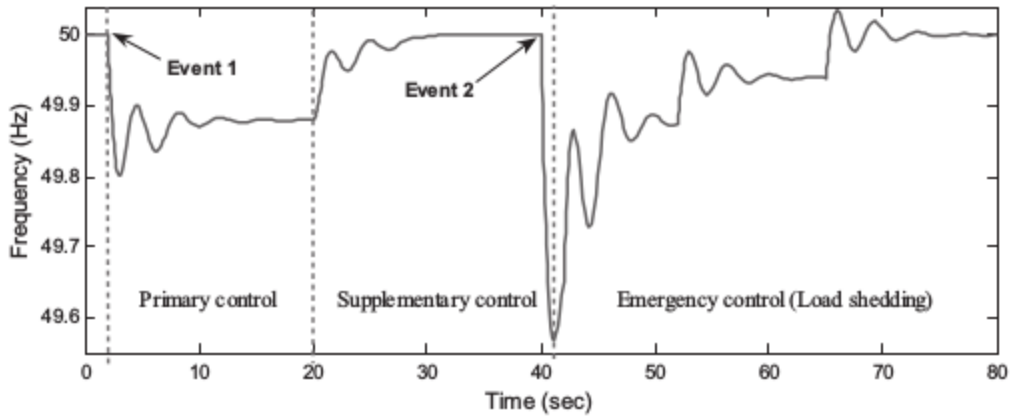

Figure 2.4: A minor frequency disturbance occurs at the 2 sec mark and primary control systems (governors) arrest the frequency drop. System frequency is adjusted to desired 50Hz operating level using AGC control of regulating units. This referred to as supplementary (or secondary) control in the literature. AEMO refers to this as load following ancilliary services. 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. Researchers often find it useful to divide generation assets into sub-groups 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.

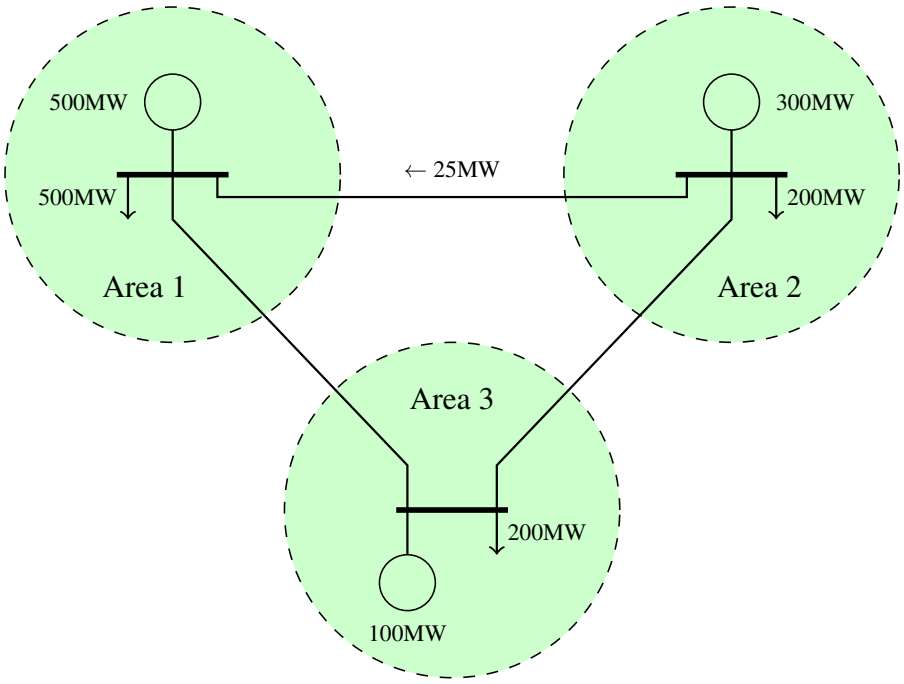

Figure 2.5: An example of three interconnected control areas in a 60Hz power system. The interconnections allow power to flow from one area to another, allowing generators to service loads from different areas. Each control area consists of several generators and loads, but are modelled with a single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

Typically, for each control area, researchers will aggregate many loads into a single load, and many generators into a single generator. This simplifies the model further [14]. The simplest 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.

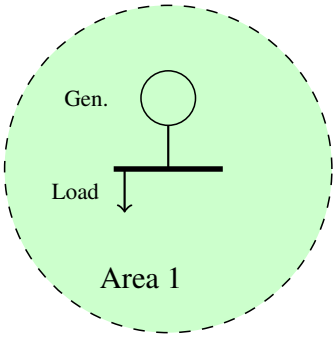

Figure 2.6: A single area power system has a load and a generator, with no connections to other power areas

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 (REFERENCE), 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 (REFERENCE). Older governors operated by mechanically coupling the governor with the rotating shaft of the generator (REFERENCE); however, newer governors operate electronically (REFERENCE). 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 (REFERENCE). 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 (REFERENCE).

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

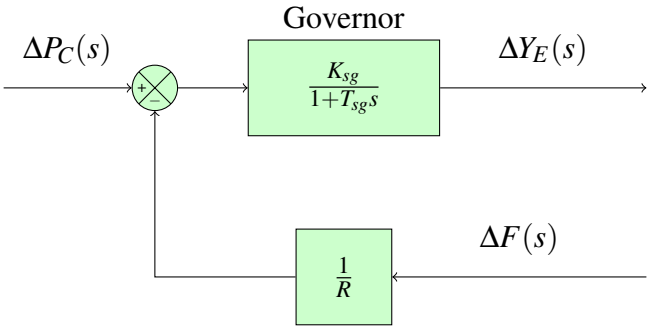

Figure 2.7: Frequency domain model for the generator governor.

Turbine Model

A steam turbine converts energy stored in the form of high pressure and high temperature steam into rotational energy (REFERENCE). The steam is created in a boiler heated by fossil fuel combustion (REFERENCE). Alternatively, a nuclear reactor could be used to produce the heat (REFERENCE). The steam is forced through turbine nozzle sections accelerating it 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 (REFERENCE).

Steam turbines come in a variety of configurations. Multiple turbines, and steam reheating systems are common features.

The second element is the turbine, shown in Figure 2.8. The frequency domain derivation of the governor model can be found in Appendix A.2.

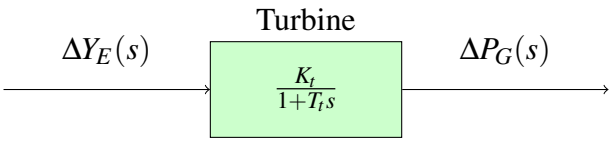

Figure 2.8: Frequency domain model for the generator turbine.

Generator Load Model

The third element is the generator-load, shown in Figure 2.9. The frequency domain derivation of the governor model can be found in Appendix A.3.

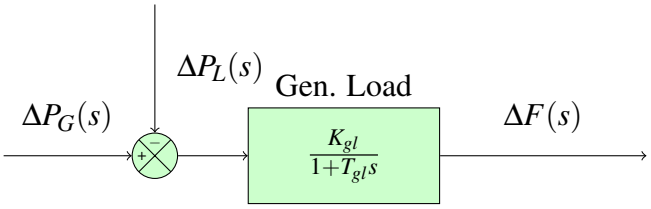

Figure 2.9: Frequency domain model for the generator-load.

2.1.2 Frequency Control for a Single Area System

It is generally acknowledged that a speed droop governor feedback control regime will perform primary frequency control, and an AGC feedback loop is 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 particular model provides a model for 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.

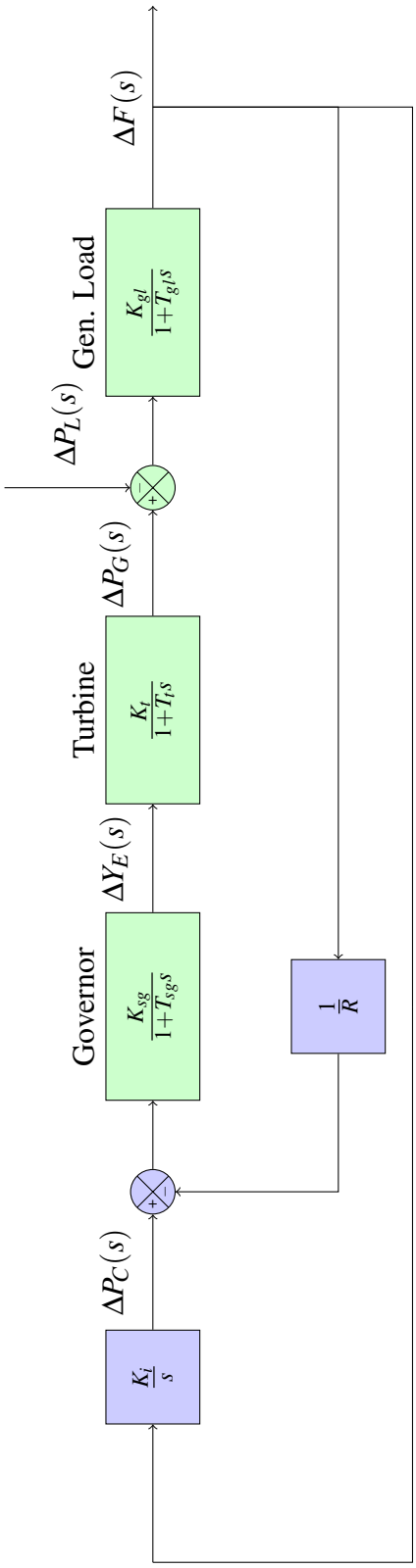

Figure 2.10: A classical feedback control approach for a single control area power system. The system is comprised of a first order models for both turbines, and generators. The governor controllers are also first order models. AGC is implemented using an integral control block in a feedback loop [17].

2.1.3 Modelling a Two Area System

The single area system presented in Section 2.1.2 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 in the literature as tie lines). Often it is the case that there is some net power transfer over the tie lines, enforceable by economic contract. Single area models do not provide for this additional complexity.

Distinct control areas are typically thought of as different participants in the generation market, or simply as different regions in which generation assets are based [13].

The simplest model that includes tie lines is the two area power system, shown in Figure 2.11.

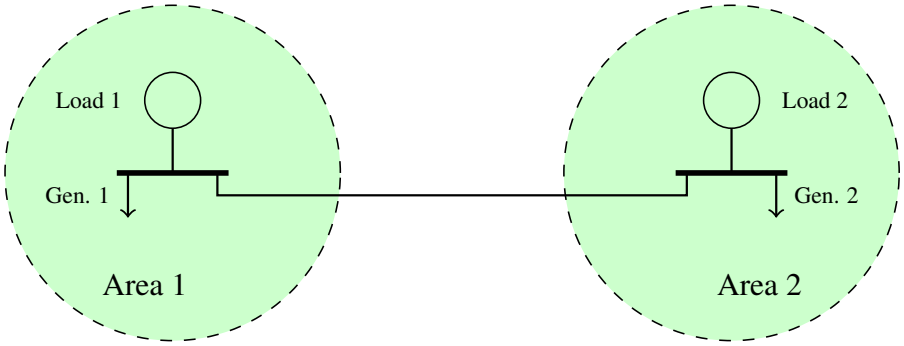

Figure 2.11: A two area power system comprised of generators and load connected via a tie line. Power flows from one area to the other depending on the power demands.

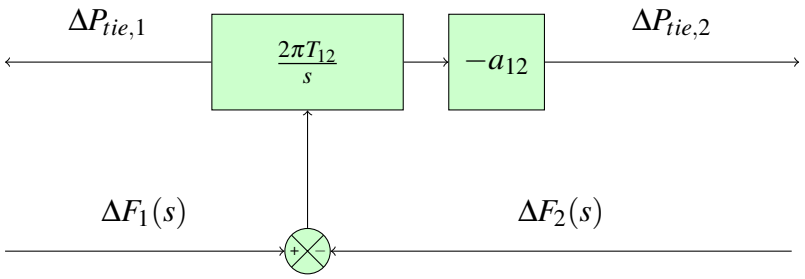

Figure 2.12: Frequency domain model for the transmission line between two power areas.

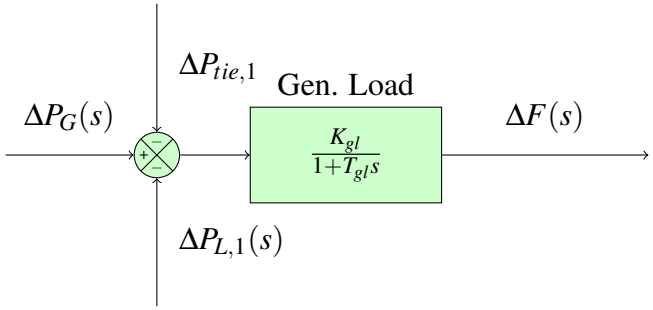

Figure 2.13: Frequency domain model for the generator-load with feedback from a tie-line.

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. Linear models have been developed to simulate these systems, and classical control approaches have been successfully implemented to meet the new control objectives. In order to achieve this, 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 the . The implementation of this control system is shown in Figure 2.14.

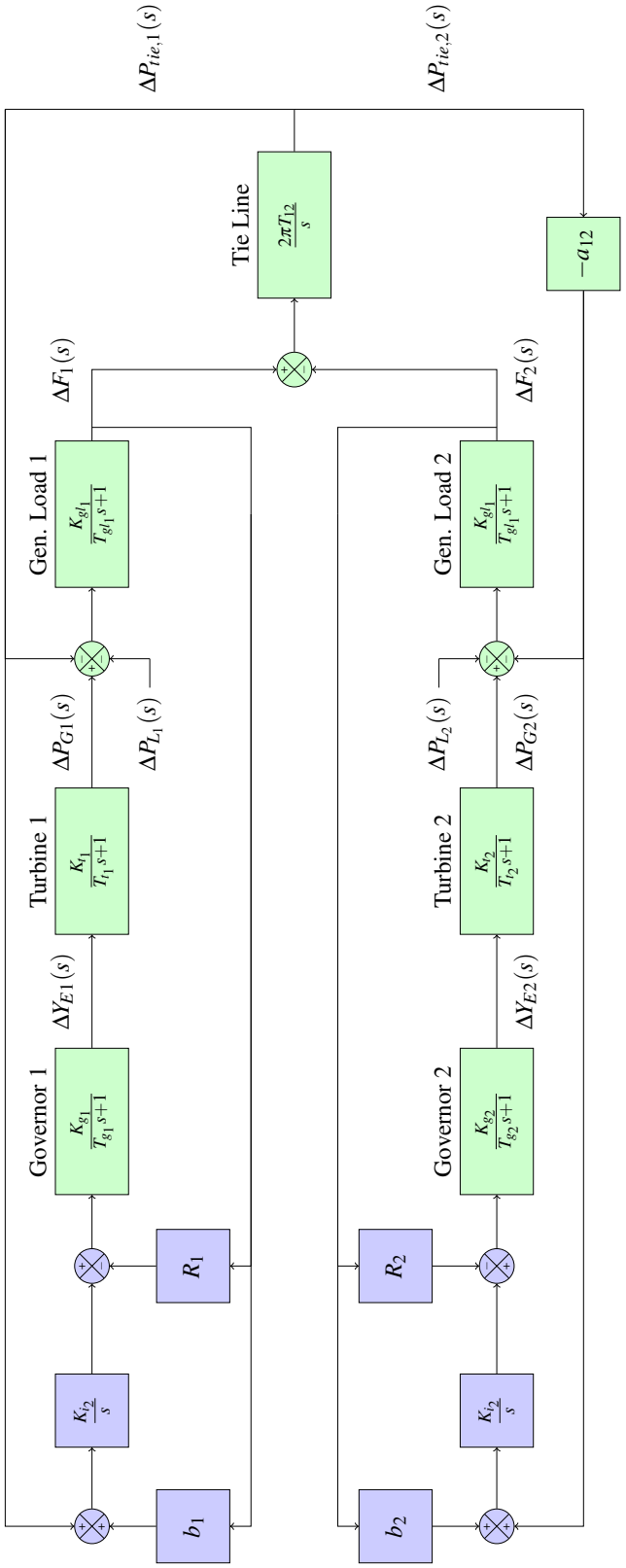

Figure 2.14: A classical feedback control approach to a two area power system [17].

2.2 Reinforcement Learning

According to Sutton and Barto's seminal text, reinforcement learning (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 it's 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 diagramatically in Figure 2.15.

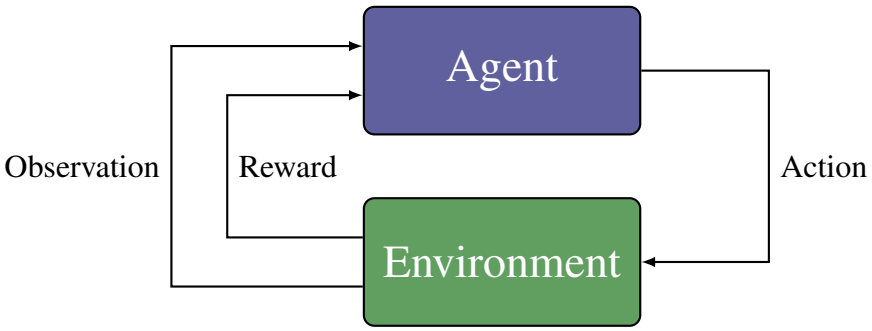

Figure 2.15: Reinforcement learning can be thought of as two discrete steps: action and observation. The agent takes an action and receives a reward. The action causes the agent to change state in the environment.

One of the most common ways to represent the RL problem is to model the environment 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 which 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 the Markov Decision Process (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 $R : S \times A \times S \to [R_{min}, R_{max}]$. Fundamental to Bellman's MDPs were the state transition dynamics which 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 yield reward, $r \in R$, with some given probability. This set of probabilities are assigned by

a state transition function $P : S \times A \rightarrow S$. Generally, the a single reward is bound to a state transition so function P can be thought to assign a state and reward. The function P, and it's simpler notation p, is typically written as

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

The set of parameters outlined above, and expression 2.2, make up a framework referred to as an MDP.

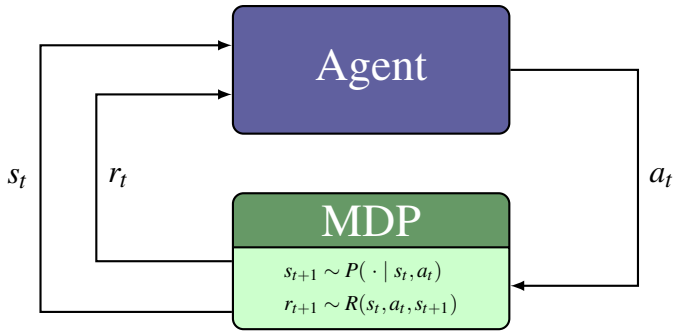

Figure 2.16: Reinforcement learning using an MDP architecture to model state transitions in the environment

2.2.2 Returns, Episodes, and Policy

In addition to developing the MDP framework, Bellman was also responsible for key developments in a field of research called 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 it's cumulative reward [20], [21]. This idea was originally thought to be distinct from RL. The main difference is that DP provides the agent with complete knowledge of it's environment, whereas RL agents have no knowledge of environment dynamics and must learn them as well as how to maximise their cumulative reward [18]. Many researchers made links between DP and RL [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 for use with RL problems [25]. Central ideas used in DP algorithms include episodes, returns, and policies [18].

The set of states, action, 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 as it concludes 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 it's current state. This belief is called a policy and denoted as 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 $\pi*$, 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 useful 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 was useful to Bellman because it provided a way to check if one policy was 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 can not be used to specify the policy it evaluates. Bellman developed a variation of equation 2.5 called the *state-action value function* that can evaluate and specify a policy. The state-action value function, $Q_\pi(s,a)$, is defined as the expected sum of the discounted return, G_t, that 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 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 opitmality 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 a high level overview of value function based methods. The most common value function based methods are Monte Carlo and temporal difference approaches.

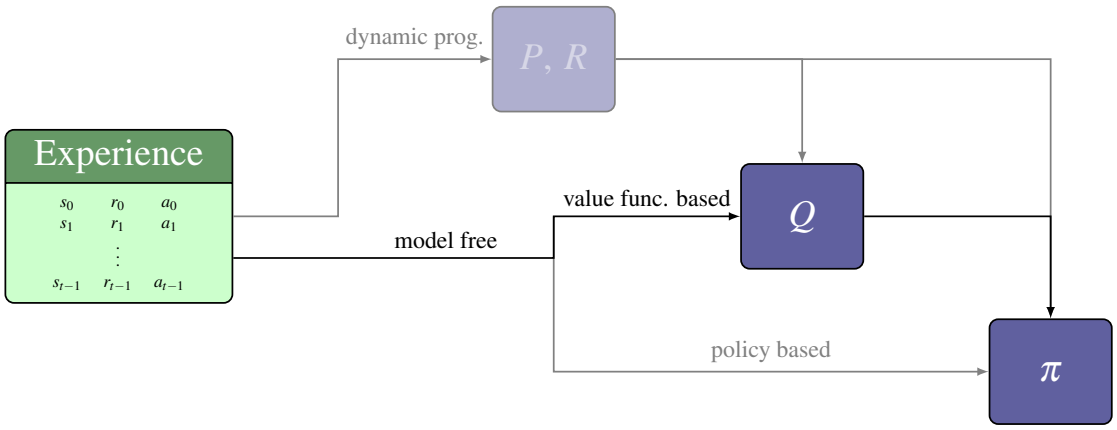

Figure 2.17: Value based approaches try to estimate the state-action value function

Monte Carlo Methods

Sutton and Barto were the first to introduce a version of the Monte Carlo (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 (2.9) from the current iteration of the state-action value function, then MC control would not always converge to the optimal state-action

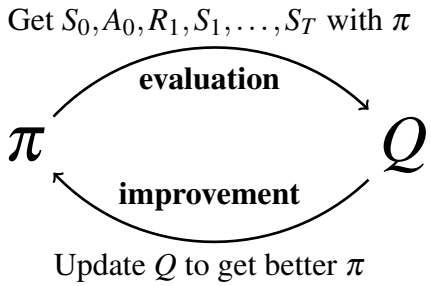

Figure 2.18: The Monte Carlo algorithm takes a policy evaluation step and policy improvement step, for each episodic iteration.

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 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.

The most common method of overcoming this problem is to use an ε-greedy policy instead of the greedy policy. The basic 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. The update value was then multiplied by some value $\alpha \in [0,1]$ to ensure that the update steps were 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 Constant α Monte Carlo control algorithm. The algorithm is shown in listing 1. Note that the algorithm returns as approx-

imation 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 (MC) 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 a high level overview of policy search methods compared to dynamic programming and value function based methods.

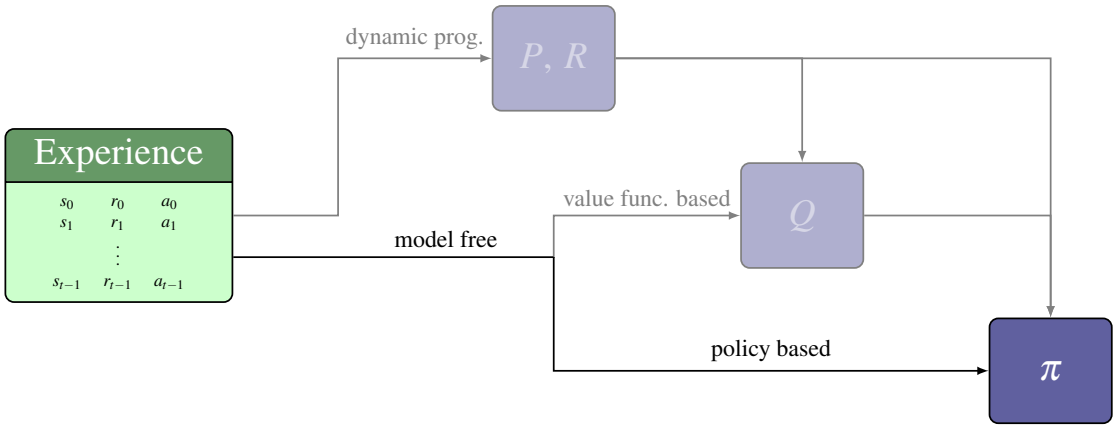

Figure 2.19: Policy search approaches try to estimate the policy directly without using state-action values.

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.

The basic idea behind policy search is that 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 well known algorithm that is used to solve 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 has better convergence properties and can learn stochastic policies which are not possible with value based approaches

NEED TO CONCLUDE THIS SECTION

2.3 Deep Neural Networks

Deep neural networks are the technology responsible for some of the most recent state-of-the-art technological breakthroughs in fields such as audio to text speech recognition systems [26], image classification systems [27]–[30], text-to-text machine translation (REFERENCE), and robotics [31]–[34].

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.

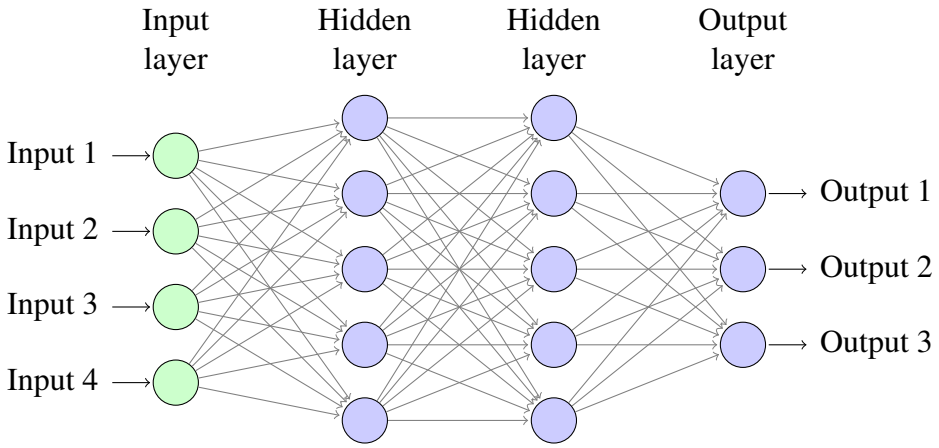

Figure 2.20: An example of a feedforward network consisting of an input layer, two hidden layers, and an output layer

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 [35]. 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" [36]. Rosenblatt's perceptron model consisted of a single node used for binary classification of patterns that

are linearly separable [37]. 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 provides an shows the computational model of a neuron, expressing 2.21.

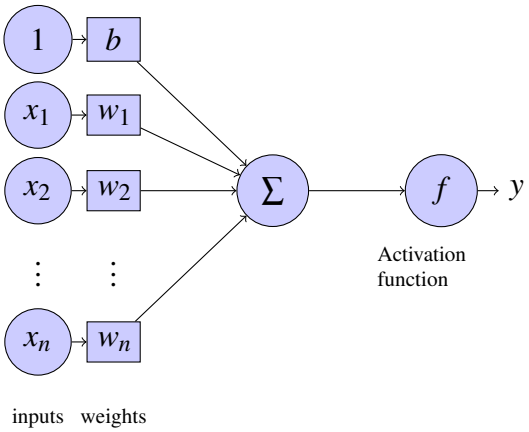

Figure 2.21: Rosenblatt's perceptron model passes the summation of weighted inputs to an activation function

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 [38]. This limitation was overcome by increasing the size of neural networks, which subsequently required the development of new algorithms to train the networks. One such algorithm is backpropagation [39]. The backpropagation algorithm is discussed further in §2.3.4; however, one important aspect is that it requires a differentiable activation function (REFERENCE). The Heaviside function is non-differentiable at $x = 0$, and has a zero derivative elsewhere and is therefore not suitable (REFERENCE). According to Haykin, a number of different activation functions can be adopted to replace the Heaviside function [35]. 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

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

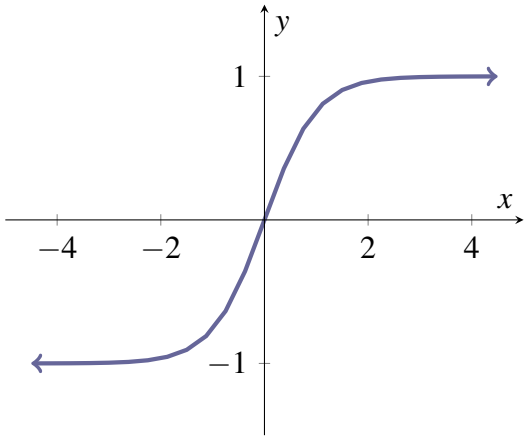

Figure 2.22: Hyperbolic tangent activation function

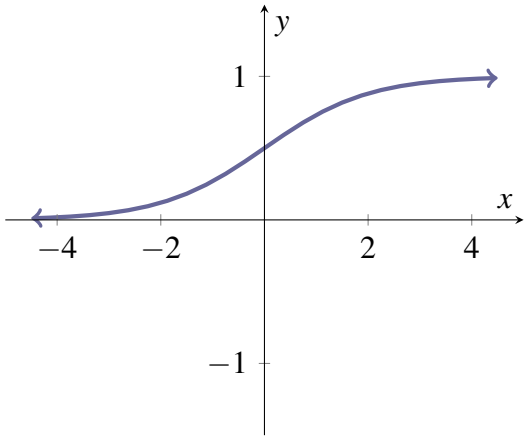

Figure 2.23: Sigmoid activation function

For shallow neural networks, the hyperbolic tangent and sigmoid activation functions work well. 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 [40]. It describes a situation where the gradients for hyperbolic tangent and sigmoid activation functions become close to zero toward the function tails. The result is that the backpropagation algorithm sends very small error signals general inability 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 $\alpha < 1$

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

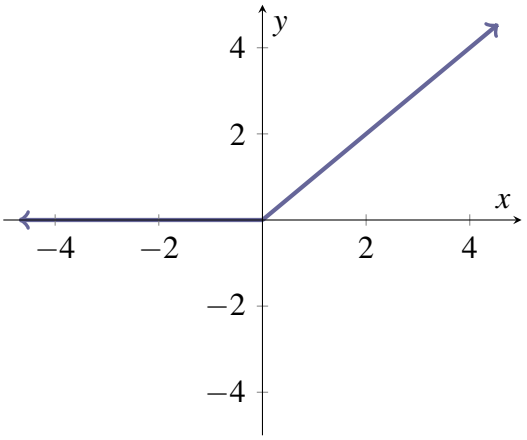

Figure 2.24: ReLU activation function

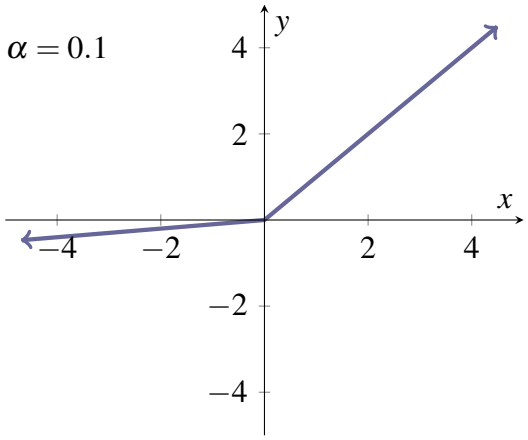

Figure 2.25: LReLU activation function

what is interesting about the relu is that theoretically they should not work with back-propagation since

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 (REFERENCE), which minimises a cost function (REFERENCE). 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 (REFERENCE). Often the Mean Squared Error is used as a cost function, although cost function specification is dependent on the neural network application (REFERENCE). 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 (REFERENCE).

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

talk about stochastic gradient descent

talk about the adam algorithm

werbos backpropagation

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 a ground breaking paper that presented a novel agent: a deep Q-network (DQN) [31]. 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. Adapting the Q-learning algorithm, Mnih *et alias* 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 level performance across a set of 49 Atari 2600 games.

Historically, applications of neural networks to reinforcement learning failed because they were unstable and divergent. One of the reasons was due to temporally correlated sequences of states, actions, and rewards arising during an episode causing over fitting

in the network during training [41]. Mnih *et alias* 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 $\hat{Q}(s,a;\theta^-)$. 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 update (2.16):

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

Appling gradient descent to 2.27 creates error values which are backpropagated through the network to adjust 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

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

2.4.2 Deep Deterministic Policy Gradient

In late 2015 Lillicrap *et alias* adapted key ideas from DQN and the deterministic policy gradient theorem from [42] to develop an algorithm, Deep Deterministic Policy Gradient (DDPG), that could train a neural network successfully on problems with continuous action domains [32]. DDPG uses a total of four neural networks. The first network is called the actor, denoted by π. The second network is called the critic, denoted by Q. DDPG makes use of DQN's target networks idea, implementing a further two neural networks — one for each of the actor and critic networks.

Algorithm 4 DDPG Algorithm

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

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

15: Update the target networks:

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

16: **end for**
17: **end for**

Chapter 3

Literature Review

Power systems are non-linear; however, traditional control structures for maintaining power system frequency such as load frequency control (LFC) or automatic generation control (AGC), are designed under the assumption that plant modelled using linear ordinary differential equations can capture the dynamic behaviour of existing power systems. If frequency deviations are small, then this assumption is reasonable for the amount of non-linearity present in existing power systems. If frequency deviations are large, or additional non-linearity were introduced to the system, then it is no longer reasonable to assume linear ODE system models.

Owing to an increase in the proportion of photovoltaic power generation, along with an increase in the use of high voltage direct current (HVDC) transmission lines in Australia's power network, power system dynamics are becoming more non-linear (REFERENCE). This is creating a need to explore novel control architectures for AGC in order to improve control performance for the non-linear system. One such control architecture being investigated is deep reinforcement Learning (DRL). To fully grasp this problem two key areas of understanding are necessary. Firstly, it is important to know what LFC is and the control architectures that have previously been explored to address this problem. Secondly, knowledge of DRL and its historical applications to control problems is required to understand strengths, limitations, and underlying assumptions of the architecture.

3.1 Classical Controllers

Since Thomas Edison's first commercial power station, commissioned in 1882 at 255-257 Pearl Street New York, controlling power frequency has been a key factor in power generation [43]. One of the first attempts to control the frequency for a single generator and load system was with a device called a turbine governor (REFERENCE). The turbine governor was designed with an in-built proportional feedback control loop (REFERENCE). DESCRIBE HOW THIS ACTUALLY WORKS. The literature often refers

to this as the primary control loop [16]. Operators often found that primary loop governor control would require constant fine tuning to ensure that the power system was operating at the scheduled frequency (REFERENCE). XXXX undertook a mathematical analysis using first order linear models for the governor, turbine, and generation load control (REFERENCE). The analysis showed that primary control with a governor was successful in arresting frequency deviations from the desired set point, but persistent offset errors from the set point prevented the uptake of the technology [44]. Later research concluded that a secondary control loop to the governor was required to provide sufficient frequency control [45]. The secondary control loop provided integral control (REFERENCE). Mathematical analysis, using first order linear models for plant showed that atuned proportional-integral (PI) controller was able to arrest frequency deviations and subsequently restore the system frequency to it's scheduled value. The PI control scheme constitutes the classical approach to the solving the LFC problem (REFERENCE).

Cohn [46] and Aggarwal *et al.* [47], [48] undertook pioneering work to develop classical control approaches to work with power systems comprised of two or more control areas. A system made up of more than one power system area required frequency control, but also the control of the power flow over the transmission infrastructure (tie lines) which connected the areas (REFERENCE). 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 (REFERENCE). One of the most important things to come out of his work was 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 deviations, and that unscheduled tie line power flows were minimised between neighbouring control areas [49]. Classical power system frequency controllers can be designed using Bode and Nyquist diagrams to obtain desired gain and phase margins. Root locus plots can also be used [50]. 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. This is especially true in the presence of parameter variations and nonlinearities [17], [45], [51].

Frequency controller analysis and design assumes plant models have linear dynamics; however, studies have shown that modern power systems display complex non-linear dynamics [52]–[55]. Modern power systems are large-scale and comprised of multiple power generation sources such as thermal, hydro, and photovoltaic power — some of the more commonly researched generator non-linearity include governor dead band (GDB) [52] and generator ramp constraint (GRC) [53], [54]. 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

(REFERENCE).

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 (REFERENCE). 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 [55]. In recent years, frequency control methods using fuzzy logic, genetic algorithms (GAs), and artificial neural networks (ANNs), have attempted way 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. Moreover, it was found fuzzy controllers are simpler to implement [56], [57]. Yesil et al. [58] 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

Genetric 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 which 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. An individuals chance of being selected for used in an genetic operation is based on an objective measure of fitness — strong individuals are retained and weak individuals are discarded [59].

Chang et al. [60] 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. [61] took this one step further by optimally tuning 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 a higher order control system. Research undertaken by Ghosal [62] 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 Artificial Neural Networks

Artificial neural networks are systems that take input signals and, using many simple processing elements, produce output signals. The processing elements, or neurons, each have a number of internal parameters referred to as weights. Changing a weight will change the behaviour of a neuron. If many weights are changed, the behaviour of the ANN can be changed. The goal is to choose weights of the network in order to achieve the desired input/output relationship — this is called training the network [63].

Beaufays et al. [64] demonstrated it was possible to used a neural network for frequency control in one and two-area power systems. The ANN replaced the integral controller in the classical structure; however, employed a state variable vector input containing frequency deviation and tie-line power measurements instead of a single value ACE signal seen with classical controllers. The network was trained using a back propagation through time algorithm, and resulted in better transient performance when compared with a classical PI controller. Using these results, Demiroren et al. [65] 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. Research undertaken a year later confirmed these results for a larger four-area power system with thermal and hydro generation sources [66].

3.5 Deep Reinforcement Learning

Chapter 4

Approach

The approach is broken up into four phases, undertaken sequentially. The first phase focuses on model development and implementation of the power system environment, proportional integral controller, and the deep deterministic policy gradient (DDPG) agent. The second phase undertakes preliminary investigations of simulation system settings and DDPG hyperparameters to understand DDPG power system control feasibility. The third phase conducts experiments on both the DDPG and PI controller capacity to cope with stochastic load demand profiles, and non-linear plant. Finally, the fourth phase will analyse the results and compare the performance of DDPG against PI controllers.

4.1 Model Development and Implementation

Model development is focused on obtaining the correct 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; however, also reviews Python class implementations for the environment, PI controllers, and neural networks.

4.1.1 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. The main reason for taking this approach is to provide 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 a common practice when developing environments for reinforcement learning [67]. Additionally, frequency domain simulation techniques, such as Laplacian transforms, have strict initial condition assumptions [50], that would limit the reach of research findings to real world applications.

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

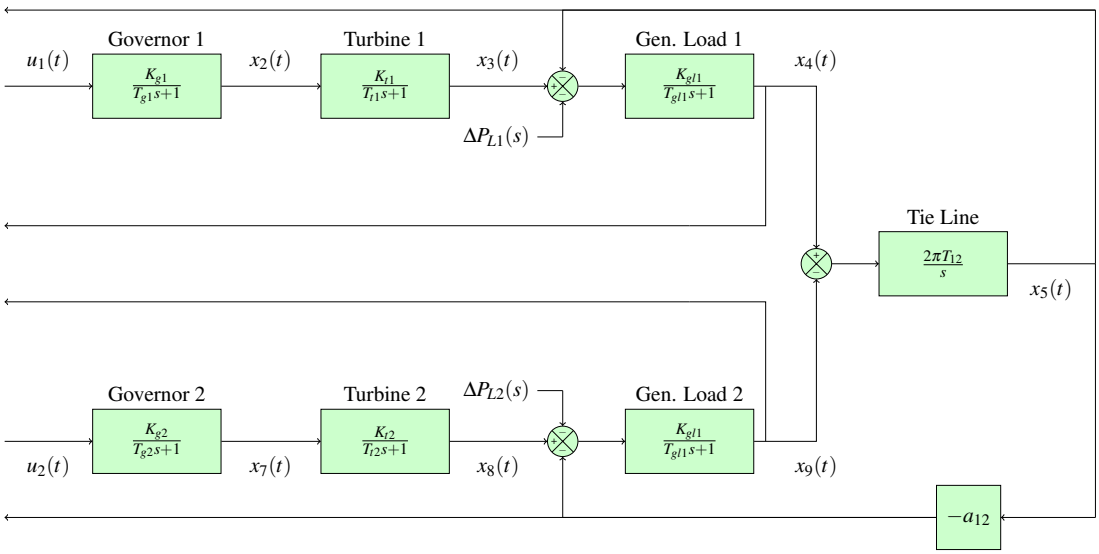

Figure 4.1: Variable assignment for a two area power system environment to model in the temporal domain

Suppose variables are assigned to the power system according to Figure 4.1. The first order system of linear ordinary differential equations for area 1 is:

$$\dot{x}_2(t) = \frac{1}{T_{sg_1}}\left(K_{sg_1}u_1(t) - x_2(t)\right) \tag{4.1}$$

$$\dot{x}_3(t) = \frac{1}{T_{t_1}}\left(K_{t_1}x_2(t) - x_3(t)\right) \tag{4.2}$$

$$\dot{x}_4(t) = \frac{1}{T_{gl_1}}\left(K_{gl_1}\left(x_3(t) - x_5(t) - \Delta p_{L1}(t)\right) - x_4(t)\right) \tag{4.3}$$

$$\dot{x}_5(t) = 2\pi T_{12}\left(x_4(t) - x_9(t)\right) \tag{4.4}$$

$$\dot{x}_7(t) = \frac{1}{T_{sg_2}}\left(K_{sg_2}u_2(t) - x_7(t)\right) \tag{4.5}$$

$$\dot{x}_8(t) = \frac{1}{T_{t_2}}\left(K_{t_2}x_7(t) - x_8(t)\right) \tag{4.6}$$

$$\dot{x}_9(t) = \frac{1}{T_{gl_2}}\left(K_{gl_2}\left(x_8(t) - x_5(t) - \Delta p_{L2}(t)\right) - x_9(t)\right) \tag{4.7}$$

A full derivation of the first order linear system described by equations 4.1 through 4.7 is described in Appendix C.1. The system of equation 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.

4.1.2 Classical PI Controller Model

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

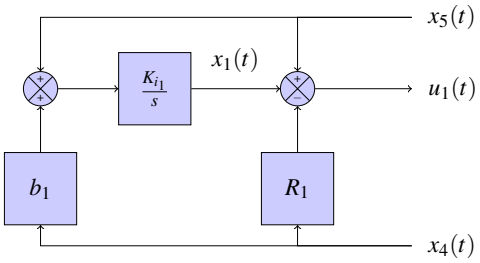

Figure 4.2: Variable assignment for the PI controller for area 1 in order to model in the temporal domain

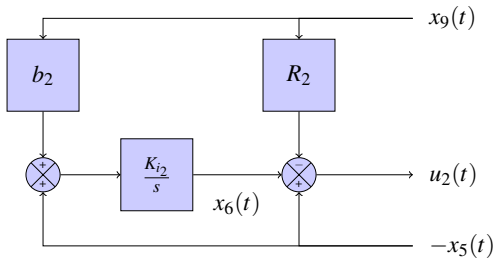

Figure 4.3: Variable assignment for the PI controller for area 2 in order to model in the temporal domain

Suppose variables are assigned to the controller for area 1 and the controller for area 2 according to Figures 4.2 and 4.3, respectively. The first order system of linear differential equations for the PI controller are:

$$\dot{x}_1(t) = b_1 \Delta f_1(t) + x_5(t) \tag{4.8}$$

$$\dot{x}_6(t) = b_2 \Delta f_2(t) - x_5(t) \tag{4.9}$$

Note that once the PI controller has been 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:

$$u_1(t) = x_1(t) + x_5(t) - R_1 x_4(t) \tag{4.10}$$

$$u_2(t) = x_6(t) - x_5(t) - R_2 x_9(t) \tag{4.11}$$

A full derivation of the first order linear system described by equations 4.8 and 4.9 is described in Appendix C.2. The system of equations were 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.

4.1.3 DDPG Controller

Neural Networks

Deep deterministic policy gradient (DDPG), described in §2.4.2, is used to train neural networks to perform control actions given a state observation. Recall that 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 alias* in their work with robotics. 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. Both actor and critic architectures contain 121800 weight parameters. The details of the actor and critic networks are shown in Table 4.2.

Table 4.1: 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

Alternative neural network architectures were also developed. These alternative actor and critic architectures occur frequently in the literature for DDPG control of robotic systems and contain less weight parameters than the networks used by Lillicrap *et alias*. The actor network contains a total of 1280 weight parameters, and the critic network contains a total of 99200 weight parameters. An additional feature of the critic is that it makes use of Leaky ReLUs to avoid the dying ReLU problem discussed in §2.3.3. Implementations for these alternative classes can be found in Appendices D.5 and D.6 for the actor and critic, respectively. Network details can be found in Table 4.2.

Table 4.2: An overview of actor and critic neural network architectures seen frequently in the literature for deep reinforcement learning for robotic system control

Layer	Actor		Critic	
	Activation Function	**Number of Perceptrons**	**Activation Function**	**Number of Perceptrons**
Input Layer	None	3	None	3
Hidden Layer 1	ReLU	256	LReLU	256
Hidden Layer 2			LReLU	256
Hidden Layer 3			LReLU	128
Output Layer	tanh	2	None	1

Training

Actor and critic neural networks are trained using the DDPG algorithm, shown in listing 2.4.2. 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 important methods used for model training. A description of key methods is provided in Table 4.3. The `DdpgController` class implementation can be found in Appendix D.7.

Table 4.3: 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 as input for a call to the neural network method `forward` which returns an action
`learn`	Performs gradient descent on the actor and critic loss functions and backpropagates errors to adjust weights for each respective network for set of random uniformly sampled experiences from the experience replay buffer

The function `ddpg_train` runs 200 episodes, each with 3000 time steps of magnitude 0.01 sec.

4.1.4 Simulation

Method implementation details are shown in Appendix XXXX. The method is used as an argument for another method of the same class, called `step`. When `step` is called with `int_power_system_sim` as an argument, the system simulation will iterate a single time step, which is performed using function `odeint` from Python's Scipy library.

Policy network architecture can significantly impact results for DDPG. Furthermore, certain activation functions such as rectified linear units (ReLU) have been shown to cause worsened learning performance due to the dying relu problem.

4.2 Preliminary Investigation

As discussed in §XXXX, DDPG performance has been observed to suffer from instability and variance, affecting the agent's ability to learn. Some of the main causes of poor agent performance are: poorly specified reward functions,

basically need to talk about how a two area power system model was found with a tuned PI controller - the parameters need to be specified and then systematic tests training the ddpg controller can be undertaken and compared against the pi controller.

reward function, network selection, hyperparameters, control signal clipping,

need to ensure that the pi controller is tuned

4.2.1 Reward Function

4.2.2 DDPG Neural Network Selection

4.2.3 Training Hyperparameters

Tuned hyperparameters play a large role in eliciting the best results from DDPG

4.2.4 Ornstein–Uhlenbeck Process

4.3 Experimentation

4.3.1 Modelling Non-linearity

4.3.2 Real World Load Demand

4.4 Analysis

Chapter 5

Preliminary Investigation

Chapter 6

Analysis and Discussion of Results

Chapter 7

Conclusion

Chapter 8

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] 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.

[27] 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.

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

[29] 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.

[30] 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.

[31] 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.

[32] 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].

[33] 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.

[34] 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].

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

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

[37] ——, "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.

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

[39] 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.

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

[41] 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.

[42] 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, 22–24 Jun 2014, pp. 387–395. [Online]. Available: http://proceedings.mlr.press/v32/silver14.html.

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

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

[45] 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.

[46] 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.

[47] 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.

[48] ——, "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.

[49] 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.

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

[51] 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.

[52] 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.

[53] 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/T-PAS.1975.32006. [Online]. Available: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1601608.

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

[55] 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.

[56] 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.

[57] 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.

[58] 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.

[59] 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.

[60] 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.

[61] 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.

[62] 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.

[63] D. H. Nguyen and B. Widrow, "Neural networks for self-learning control systems," *IEEE Control Systems Magazine*, vol. 10, no. 3, pp. 18–23, Apr. 1990, ISSN: 2374-9385. DOI: 10.1109/37.55119.

[64] 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.

[65] 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.

[66] 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.

[67] 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 looking at a single area power system, there are three main components that the literature has a tendency to focus 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; and
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 (the pair of balls) which spin around a central axis. These masses are mechanically coupled to the the turbine drive shaft, so their angular velocity is a function of the turbine speed. Elgerd and Fosha's text [45] provides a really great schematic representation of the governing system for a steam turbine, shown in Figure A.1. This schematic is used to derive the plant model for the governor.

If we let A on the in the schematic be moved downward a little bit, Δy_A, the turbine power output will change by a directly proportional amount. Letting ΔP_C be 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 high pressure oil will flow onto the top of the main piston forcing it downwards. As the steam valve opens,

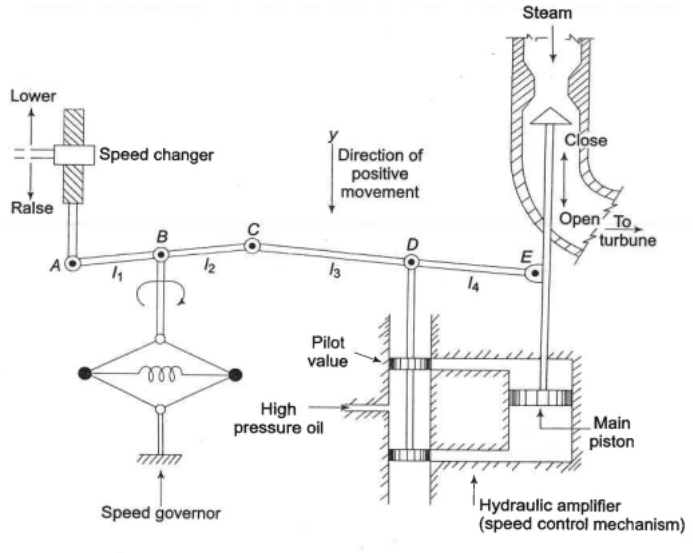

Figure A.1: A schematic of a steam governor

more steam will drive the turbine faster and causing the flyball governor to lower point B. Mathematically, the movement of C can be expressed as the result of two separate inputs:

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. Given a frequency increase Δf point B will move downward so, assuming A is fixed, then using 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 movement of point C and E, can be undertaken to mathematically express the movement of point D. The analysis also 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:

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

When there is some movement, Δy_D, of point D the ports of the pilot valve will open and high pressure oil will plow onto the cylinder causing some movement Δy_E. If point D moves up, high pressure oil will move point E down, and conversely if point D moves down, high pressure oil will move 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 by high pressure oil
2. Due to the first assumption, the rate of oil admitted to to the cylinder is proportional to the port opening Δy_D.

The volume of oil admitted to the cylinder is thus 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 gives the following:

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

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

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

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 the 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 governor model can be seen in Figure A.2 below.

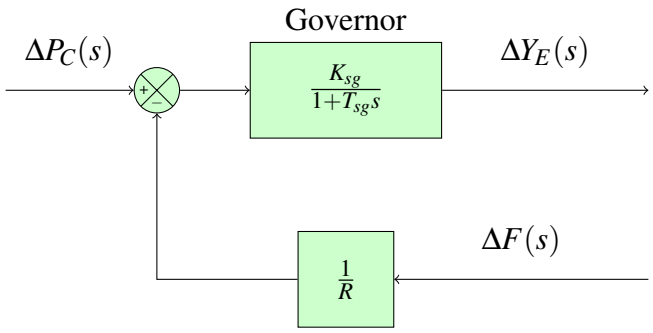

Figure A.2: Block diagram of the steam governor model in the frequency domain

A.2 Turbine Model

Consider a fossil-fuelled single reheat tandem-compound turbine. For the purposes of developing a model, a basic configuration of components can be seen in Figure 1. Steam enters the high pressure (HP) section through the control valve and the inlet piping. The housing for the control valves is called the steam chest. The HP exhaust steam is passed through the re-heater. The reheat steam flows into the IP turbine section through the reheat intercept valve (IV) and the inlet piping. The crossover piping provides a path for the steam from IP section exhaust to the low pressure (LP) inlet.

The 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 opening exhibits a time constant T_{CH} due to the charging time of the steam chest and the inlet piping to the HP section. The intercept valve is normally used for rapid control of the turbine in the event of an overspeed, and won't be considered in this analysis. The reheater holds a substantial amount of steam and has a time constant T_{RH}. The

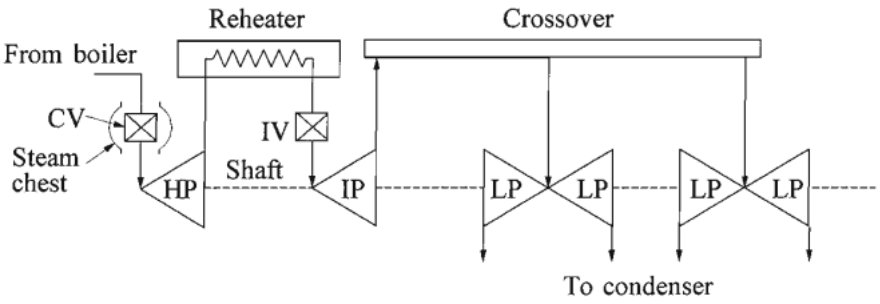

Figure A.3: Configuration of a fossil-fuelled single reheat tandem-compound turbine

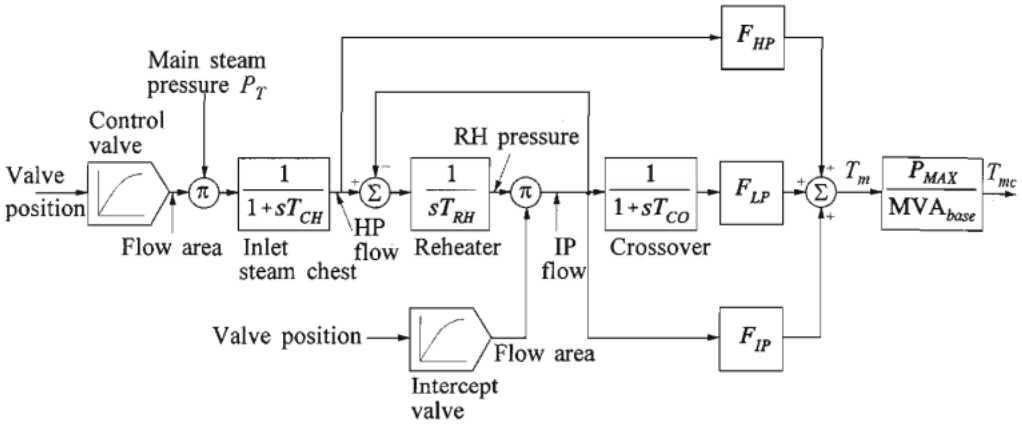

Figure A.4: Block diagram of the tandem-compound reheat turbine

steam flow into the LP sections experiences an additional time constant T_{CO} associated with the crossover piping. Figure 2 shows the block diagram representation of the tandem compound reheat turbine.

To simplify the model shown in Figure 2, 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 the 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 3 below.

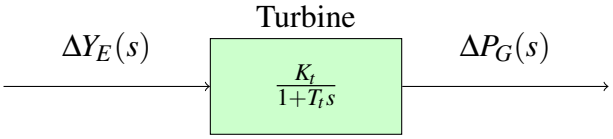

Figure A.5: Simplified block diagram of the tandem compound reheat turbine

A.3 Generator Load Model

If the turbine output is in some steady state, and the power system experiences some perturbation, then let the incremental power from the turbine be 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 it's 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:

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

The rate of change of kinetic energy is therefore:

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

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 is B is positive, then the 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 is the model of 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 XXXX below.

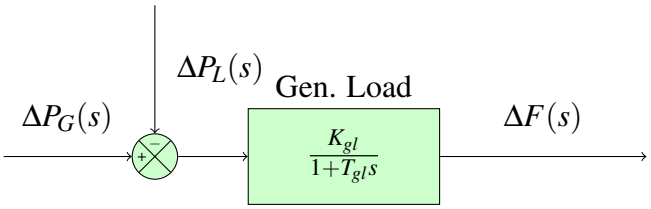

Figure A.6: Block diagram of the generator load model in the frequency domain

Appendix B

Frequency Domain Modelling for a Two Area Power System

B.1 Tie Line Model

Power systems in each of the control areas are very similar to the single area model shown in Figure 2. In fact, we don't have to change our governor, or turbine models. 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 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 (because frequency is basically the same as angular velocity). This allows us to re-express 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}, hence, we can write that:

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

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. These expressions need to be linked back to the power system model. To do this, first take the inverse Laplace transform of equation B.4 which yields:

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

Then we need to 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 provide guidance on how to structure the block equation for the tie line. 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 $-a_{12}$. The tie line block diagram is shown in Figure B.1.

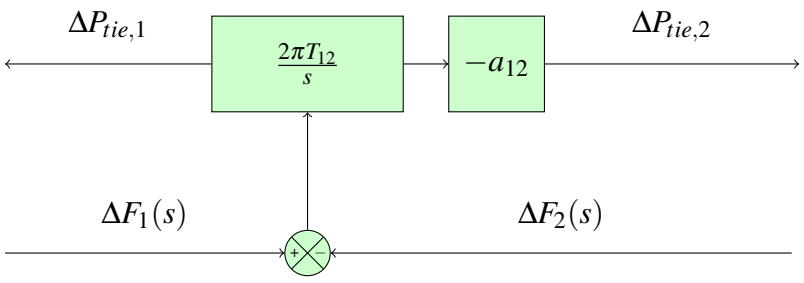

Figure B.1: A block diagram for the tie line connecting power area 1 and power area 2

B.2 Generator Load Model with Tie Line Input

The analysis in XXXX 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 given (or taken) over the tie line. Using this idea, the incremental power balance equation for area 1 can be written as:

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

A couple of parameters got introduced in equation B.9. These 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, we get:

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

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 shows that the only thing to really change is that the generator-load system now has an 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.

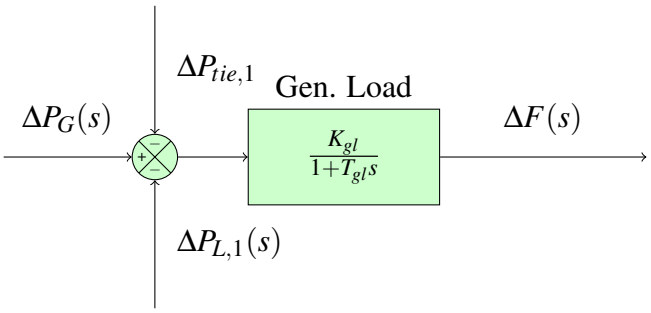

Figure B.2: Generator-load block diagram now has an additional input from the tie line connecting power area 1 and power area 2

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 **??**. 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.

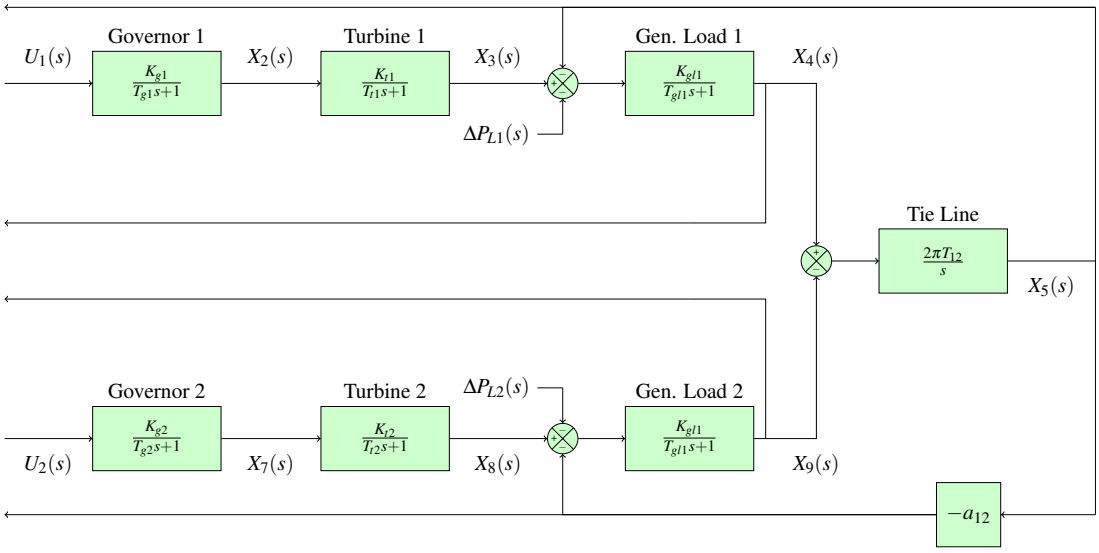

Figure C.1: The governor, turbine, generator-load, and tie-line models for both power areas.

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 **??**. 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, $X_5(s)$.

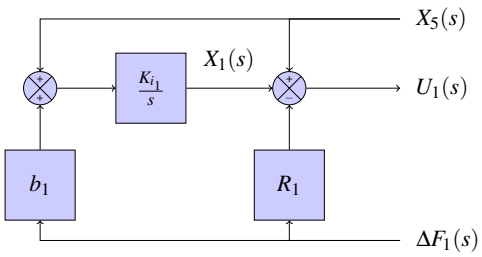

Figure C.2: Proportional integral controller for power area 1, with assigned variables for temporal domain modelling

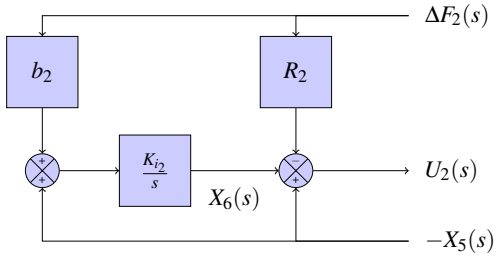

Figure C.3: Proportional integral controller for power area 2, with assigned variables for temporal domain modelling

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

        # 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)
```