UNIVERSITY
OF OSLO

Master's thesis

Deep Learning of Large Scale Biophysical Neural Models

Sebastian Amundsen

Biological and Medical Physics

60 ECTS study points

Department of Physics

Faculty of Mathematics and Natural Sciences

Spring 2023

Sebastian Amundsen

Deep Learning of Large Scale Biophysical Neural Models

Deep Learning of Large Scale Biophysical Neural Models

http://www.duo.uio.no/

Printed: Reprosentralen, University of Oslo

Abstract

In recent times, there have been developed biophysical advanced neuron models that accurately create detailed representations of the input-output relationship found in real neurons. However, these models require a lot of computational resources, which may limit their applicability for modelling large-scale neuronal systems. This thesis introduces three Multi-Task Learning (MTL) methods to address this computational challenge; the MTL methods were compared by loss and diversity metrics. We found that the Multi-gate Mixture of Experts (MMoE) and Multi-gate Mixture of Experts with Exclusivity (MMoEEx) best predicted compartmental voltage values from a biophysical Layer 5b Pyramidal Cell (L5b PC) neuron model. On the other hand, the Multi-task Hard-parameter sharing (MH) method was subpar in performance compared to the MMoE and MMoEEx. We also implemented the Loss-Balanced Task Weighting (LBTW) algorithm into our MTL methods to improve the prediction of the spiking behaviour of the neuron model; however, we did not manage to predict spiking initiation in any of our models, which is likely due to the spiking task being too different compared to the compartmental voltage values for our MTL methods to predict both simultaneously.

Acknowledgements

Initially, I would like to thank my supervisor Gaute T. Einevoll for introducing me to the field of computational neuroscience and providing a special curriculum fundamental for me to be able to approach this master thesis. I thank my supervisors, Jonas Verhellen and Kosio Beshkov, for their invaluable guidance and extensive technical support throughout the project. I would also like to thank the examiners for taking the time to read and review my master's thesis.

I want to thank UiO and the physics department; I appreciate the opportunity to pursue my education in biological physics. In addition, I would like to thank UiO for giving me access to MLNodes so I could run my simulations, and thanks to Sabry Razick for guidance on using the MLNodes cluster.

I want to thank my family back home, especially my parents, Ole Christian Amundsen and Linda Amundsen, and my brother Kristoffer Amundsen for priceless support and encouragement. Thank you to the people at the office, Marcus Berget, Kamilla Ida Julie Sulebakk, Maria Lunde Oftedahl and Karianne Strand, for providing me with laughter, joy, and insightful academic discussions. I also want to express my thanks to all my friends at Blindern Studenterhjem and the ”ulevaal duo” for giving me a wonderful student life beyond academics.

Contents

Chapter 1

Introduction

1.1 Motivation

Neuroscience is a multidisciplinary field focused on understanding how the nervous system is structured and how it functions. The advent of new technology, like advanced neural probes [1] and powerful supercomputers [2] [3] has increased our ability to delve deeper into the underlying mechanisms behind how the nervous system works. The Hodgkin and Huxley neuron model [4] is arguably the most influential neuron model to date [5]. It explains how action potentials are generated and propagated in neurons. Many of the most advanced biophysical neuron models today still use the Hodgkin and Huxley framework when modelling ionic currents [6] [7].

Complicated computational neuron models are now capable of recreating spiking behaviour observed in experimental recordings to a large degree; we will be looking at one of these models, namely a compartmental Layer 5b (L5b) Pyramidal Cell (PC) model proposed by Hay et al. [6]. The model is fitted to experimental data gathered from P36 Wistar rats. However, one of the drawbacks of such complicated biophysical neuron models is that they involve numerous computations of advanced cable equations, which are computationally expensive; this is a significant bottleneck if we wish to run these models in a reasonable amount of time. We will address this concern by implementing Multi-Task Learning (MTL) models to learn ”functions” that represent the

underlying mapping from input to output. We will employ MTL techniques when training our neural networks to replicate the compartmental voltages and spiking behaviour from the compartmental L5b PC model [6]. A similar approach has already been attempted in Beniaguev et al. [8], but they only focused on the prediction of spikes, while we will also be including compartmental voltage value predictions.

1.2 Multi-Prediction Neural Networks

We will be implementing three MTL methods with varying degrees of complexity. The simplest model is a multi-task prediction model with one convolutional network as its base; the convolutional network is connected to task specific feed forward neural networks that each predict a compartmental voltage or the spike initiation prediction. We call this the Multi-task Hard-parameter sharing (MH) model, where the bottom layer is shared between all tasks and the top layers are task specific [9]. The two other MTL models use a soft-parameter sharing mechanism that allows the tasks to train on different representations of the input data; here the base layer is composed of multiple convolutional networks that are intra-systemically connected by gate functions. The first soft-parameter MTL method is the Multi-gate Mixture-of-Experts (MMoE) model [10]. The second soft-parameter MTL method is the Multi-gate Mixture-of-Experts with Exclusivity (MMoEEx) model [11], which introduces an exclusivity mechanism to induce diversity among the convolutional networks. We will be comparing the different MTL methods to figure out the model which is best suited to predict compartmental voltages and spike initiation from the compartmental L5b PC models by Hay et al. [6]. A successful implementation of these methods can be quite impactful, as it might allow people to run larger biophysically realistic simulations.

1.3 Thesis Structure

Chapter 1 lays the foundation for the thesis by introducing the research question and establishing the context of the work in relation to neuroscience. Chapter 2 introduces the most important neuroscience concepts relevant to this thesis. We begin by looking at the underlying mechanisms behind how neurons communicate. Afterwards, we present the biophysical neuron models that produce the data on which we will train our deep neural networks (DNNs).

Chapter 3 introduces the concepts and mechanisms behind our deep learning methods. Here we go over general machine-learning theory relevant to the project and our three MTL prediction models. We also introduce a task-balancing method and address diversity metrics. The software and hardware used in this project are also highlighted in this chapter.

Chapter 4 contains the results of our simulations. Additionally, we conduct an analysis and discussion of our findings to determine the best-performing MTL method(s) by comparing weight distributions, loss values and diversity metrics. Chapter 5 concludes our thesis with a summary of our findings and suggestions for related future work.

Chapter 2

Neuroscience Background

In this chapter, we will examine some general neuroscience principles behind the compartmental L5b PC model. Much of the theory in Chapter 2 is based on popular textbooks in the field of neuroscience: chapters 2 and 3 in ”Principles of Computational Modelling in Neuroscience” by Sterrat, Graham, Gillies and Willshaw [12], chapter 1 in ”Neuronal Dynamics” by Gerstner, Kistler, Naud and Paninski [13], chapters 1, 9, 10 and 39 in ”Principles of Neural Science” by Kandel, Koester, Mack and Siegelbaum [14], and, chapters 1 and 5 in ”Neuroscience” by Purves, Augustine, Fitzpatrick, Hall, LaMantia, McNamara and Williams [15].

2.1 The Neuron

The neuron is regarded as the elementary processing unit in the brain. Neurons are connected in complex patterns, sending signals to each other. In the mammalian prefrontal cortex, we find pyramidal neurons. These pyramidal neurons comprise around 10 000 cell bodies, connected in dense networks [13]. In a human brain, we have at least 100 billion neurons [15]. A neuron is often characterised as having four distinct parts, namely the presynaptic terminals [14], the soma/cell body, the axon and the dendrites [13]; see Figure 1.

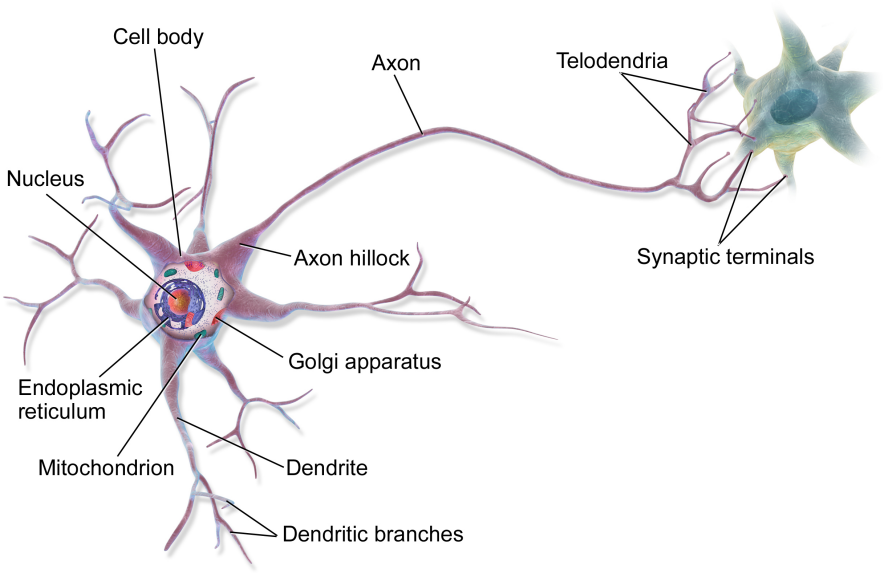

Figure 1: An illustration of a neuronal cell, with depictions of a neuron's four distinct parts: the soma, the axon, the dendrites and the presynaptic terminals. The soma is the cell body positioned between the dendrites and the axon. The nucleus is positioned inside the cell body. The axon extends out from the soma until it reaches the presynaptic terminals. The dendrites branch into tree-like structures called dendritic branches, which receive input from surrounding neurons. Reproduced from [16].

A neuron receives synaptic input at its dendrites or, more specifically, dendritic branches. The dendrites handle all incoming synaptic input from proximal neurons [13] and send the input to the cell body, where the synaptic information from multiple different dendrites is processed. In PCs, we have dendrites close and further away from the soma called basal and apical dendrites, respectively; we also have oblique dendrites positioned between the basal and apical dendrites [8]. The soma (also referred to as the cell body) is the central processing unit and the metabolic centre of the nerve cell [13] [14]. The soma integrates incoming signals from surrounding neurons, and if the input signals are ”strong” enough, an output signal is generated. The output signal travels across the axon and is sent to adjacent neuronal cells. This output signal is often referred to as an action potential. The neuron that sends the signal is called the presynaptic cell, while the receiving neuron is called the postsynaptic cell. One presynaptic neuron can connect to more than 10 000 postsynaptic neurons [13].

2.1.1 Synapses

Neuronal cells require advanced methods to interact and communicate with each other. Such communication occurs at the contacts between neurons, particularly at the synapses. We have primarily two types of synapses with fundamentally distinct transmission mechanisms; electric and chemical synapses. Electrical synapses have gap junctions that allow electrical current to move between presynaptic and postsynaptic cells; this electrical current will alter the potential of the postsynaptic membrane, affecting the generation of signal propagation in the postsynaptic cell [15].

In chemical synapses, there is a small gap known as the synaptic cleft, which separates the presynaptic and postsynaptic cell membranes. If a strong enough signal reaches the chemical synapse, it triggers a release of neurotransmitters from the presynaptic terminal into the synaptic cleft. Specific ion channels will open in the postsynaptic cell when the neurotransmitters reach the receptors in the receiving neuron. The activated ion channels will cause ions from the extracellular fluid to flow into the postsynaptic cell. This ion inflow changes the potential difference over the postsynaptic cell, eventually creating an electrical response from the chemical signal [13]; see Figure 2. The neurotransmitters sent to the receptors in the receiving neuron can lead to either excitatory or inhibitory behaviour in the postsynaptic neuron. Excitatory neurotransmitters increase the probability of signal transmission; inversely, inhibitory neurotransmitters decrease the probability of signal transmission between neurons. Whether the neurotransmitter is excitatory or inhibitory is determined by the perpetual concentration of ions inside and outside of the neuronal cell or by the ionic permeability of the ion channels affected by the neurotransmitters [15].

This thesis explores computational models based on neocortical pyramidal neurons, characterised by the soma's pyramidal shape and their apical and basal dendrites [17]. Neocortical PCs are found in the cortical column, and they work as important input-output units, which receive input from all six

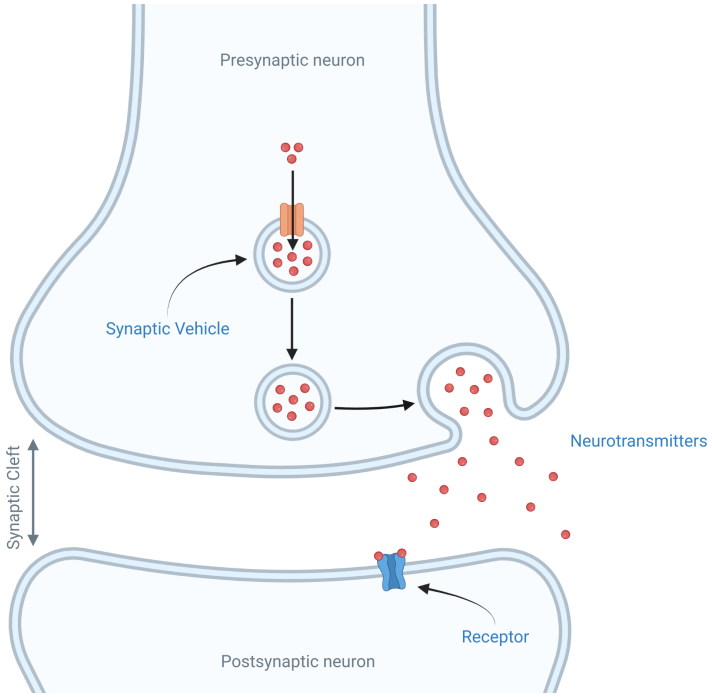

Figure 2: Illustration of a chemical synapse. The electrical signal is propagated down the axon and eventually reaches the synapses. Here neurotransmitters are released into the synaptic cleft. The neurotransmitters are then transferred to the postsynaptic neuron through receptors at the dendrites. The binding of neurotransmitters to the receptors can either lead to excitatory or inhibitory behaviour in the postsynaptic neuron. Image from Jonas Verhellen created with Biorender.com.

brain layers. This input is processed in the soma, and the related output is forwarded to other parts of the brain [6]. In Section 2.5, we will examine a model of a PC and study its response to synaptic input.

2.2 The Action Potential

It is possible to measure the voltage over the membrane of a neuron, and observations have shown that there is an electrical potential difference over the cell membrane called the membrane potential. Neurons have a resting membrane potential of around $V_m = -70mV$ when the neuron is not actively sending or receiving signals to or from other neurons [18]. The resting membrane potential V_r is a consequence of a slight separation in charge across the neuron's cell membrane, given that there are slightly more positive ions on the extracellular side and negative ions on the cytoplasmic side. The charge separation over the membrane of the cell is given by V_m:

$$V_m = V_i - V_o \tag{2.1}$$

Where V_i is the potential inside the cell, and V_o is the potential outside the cell. When the neuron is at rest, we define the outside potential V_o as zero; as a result, the resting membrane potential is $V_r = V_m = V_i$ when $V_o = 0$. There is no net movement of charge over the cell membrane if the neuronal cell is at rest; however, there is still a passive flux of ion species travelling through different classes of resting ion channels, which play a role in information transmission between neurons. The open channels in a resting nerve cell are permeable for Na^+, Cl^- and K^+ ions [14].

One of the most widespread signalling features in the nervous system is the action potential, sometimes referred to as a spike [12]. Action potentials occur when there are sequential changes in the permeability of Na^+ and K^+ over the cell membrane [14]. When membrane depolarization crosses a threshold, we obtain an action potential, where the membrane potential shoots up until it hits a peak, followed by a decrease back to the resting mem-

brane potential [12] [19]; see Figure 3.

The threshold for action potential initiation varies among different neuron types, but it is typically around $-55mV$ [20]. The shape of the action potential pulse is constant as it propagates along the axon. The amplitude of the pulse is around 100 millivolts. It is not the shape of the action potential which matters most for information transmission, but the number and timing of spikes [13]. For this reason, it is useful to focus on the timing of spikes when modelling neurons since the shape of the action potential is not the primary feature for communication between neurons. Following an action potential, a neuron will enter a refractory period, where a neuron's capacity to generate another action potential is significantly reduced. This reduction in firing capability constrains the nerve cell's firing frequency, which restricts the neurons' information transmission capability [14].

It is possible to record the brain's activity by measuring the extracellular potential around a region of brain tissue. This is done by inserting a device (like the neuropixel probe) into the brain to record contributions from spikes and ionic currents [1]. The spiking activity is extracted from the high-frequency contributions. Measurements of spiking behaviour are often combined with something called local field potentials, which we will return to in Section 2.6.

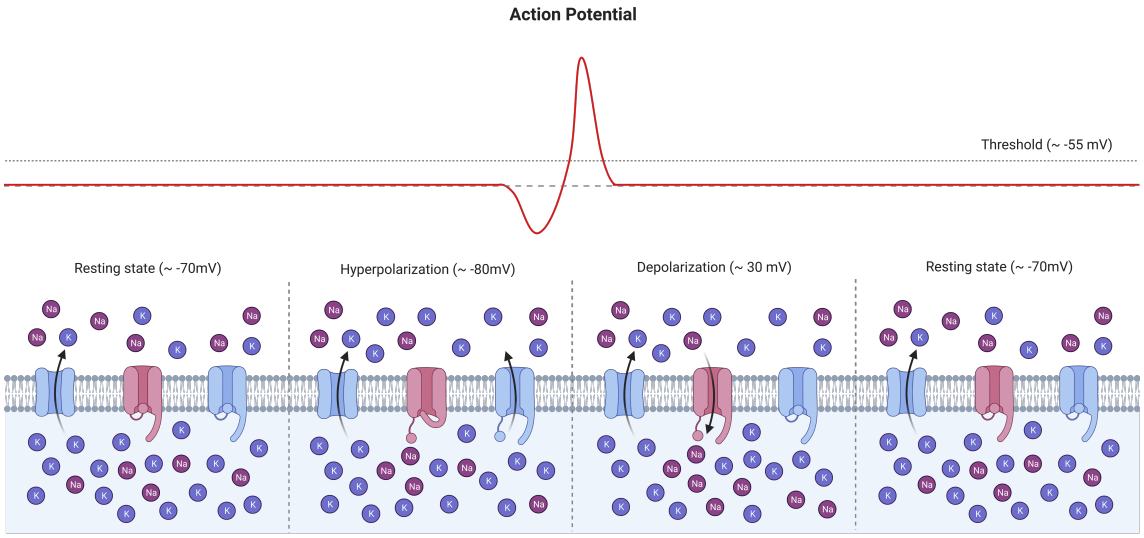

Figure 3: Depiction of an action potential. The red line illustrates the shape of the action potential, and the corresponding channel activity for each phase of the action potential is also depicted. The membrane potential "shoots up" when the membrane potential reaches a threshold of around $-55mV$; this is the depolarization phase. The membrane potential reaches a peak of around $30mV$ before it decreases back down. The refractory period is depicted as the moment when the membrane potential goes slightly under the resting membrane potential at around $-80mV$; here, the possibility of a reoccurring action potential is significantly reduced; this is the hyperpolarization phase. After some time, the membrane potential will stabilise at the resting state again, with a membrane potential at around $-70mV$. Image from Jonas Verhellen created with Biorender.com.

2.3 The Hodgkin and Huxley Model

In the early 1950s, a series of groundbreaking experiments on action potentials and ion conductances were made by Hodgkin and Huxley: "A quantitative description of membrane current and its application to conduction and excitation in nerve" [4]. They wanted to find the underlying dynamics behind the propagation of an action potential along the axon. Hodgkin and Huxley used a voltage clamp method developed by Kenneth Cole [21] to intracellularly record a giant squid axon [14]. This axon is responsible for the squid's jet propulsion and is ideal for experimentation due to its large size.

The primary ionic currents responsible for action potential initiation are

Na^+ and K^+ ions. Hodgkin and Huxley could describe the ionic currents for Na^+ and K^+ throughout the entire voltage extent during the action potential by employing the voltage clamp technique. The currents I_{Na} for Na^+ and I_K for K^+ can be expressed as:

$$\begin{aligned} I_{Na} &= g_{Na} \times (V_m - E_{Na}) \\ I_K &= g_K \times (V_m - E_k) \end{aligned} \tag{2.2}$$

Where E_{Na} is the reversal potential for Na^+, E_K is the reversal potential for K^+, g_{Na} and g_K are the conductances for Na^+ and K^+, respectively. There are also passive currents I_L that flow through resting leakage channels:

$$I_L = g_L \times (V_m - E_L) \tag{2.3}$$

Where g_L is the leak conductance and E_L is the reversal potential for the leak currents. The conductance for the leak currents g_L is constant, while sodium and potassium conductances depend on the membrane potential [12]. The conductance of the cell membrane can be described by ionic channels that permit certain ions to flow over the cell membrane. Hodgkin and Huxley represented these ionic channels as gates that can either be open or closed, where each gate has a certain number of gating particles that can be in a closed or open position. All gating particles must be open for a particular ion species to flow through the gate. We can express the conductance of potassium current g_K with the probability of a gating particle being open p_n:

$$g_k = \bar{g}_k p_n^4 \tag{2.4}$$

Where $\bar{g}_K$ is the maximum potassium conductance. The exponent of p_n relies on the number of gating particles; in our case, we have four gating particles, meaning that the exponent is four. We have a similar expression for sodium conductance:

$$g_{Na} = \bar{g}_{Na} p_m^3 p_h \tag{2.5}$$

Where $\bar{g}_{Na}$ is the maximum sodium conductance, we have two probability variables: an activation variable p_m and an inactivation variable p_h. The Hodgkin-Huxley model describes the membrane potential V_s in a small section of the giant squid axon as a differential equation:

$$C_m \frac{dV_s}{dt} = -g_L(V - E_L) - \bar{g}_{Na} p_m^3 p_h (V - E_{Na}) - \bar{g}_K p_n^4 (V - E_K) + I_c \tag{2.6}$$

Where t is time, I_c is the local circuit current, and C_m is the membrane capacitance [4] [12]. The Hodgkin and Huxley model consists of four coupled differential equations; the remaining three are for the different gate functions p_m, p_n and p_h, formulated in Appendix A. By implementing equation 2.6, we can model the membrane potential over the giant squid axon during an action potential and thus visualise the shape of the action potential. In Section 2.5, we will look at a model incorporating the Hodgkin and Huxley formalism to generate complicated compartmental L5b PC models, which generate the data used to train our DNNs.

2.4 Biophysical Neuron Models with a Detailed Morphology

Models in computational neuroscience vary a lot in complexity. Simpler models with less biological detail may be faster to simulate on a computer, but a more biologically detailed model will likely produce more accurate results. Ideally, we would like to optimise both our model's precision and the simulation's speed. In complicated neuron models, we often divide the neuron into many compartments and calculate a property for each compartment: see Figure 4. In our project, this property would be compartmental voltages and an action potential initiation prediction in response to synaptic stimuli. These properties are present in the L5b compartmental PC model [6], which we will

discuss in Section 2.5.

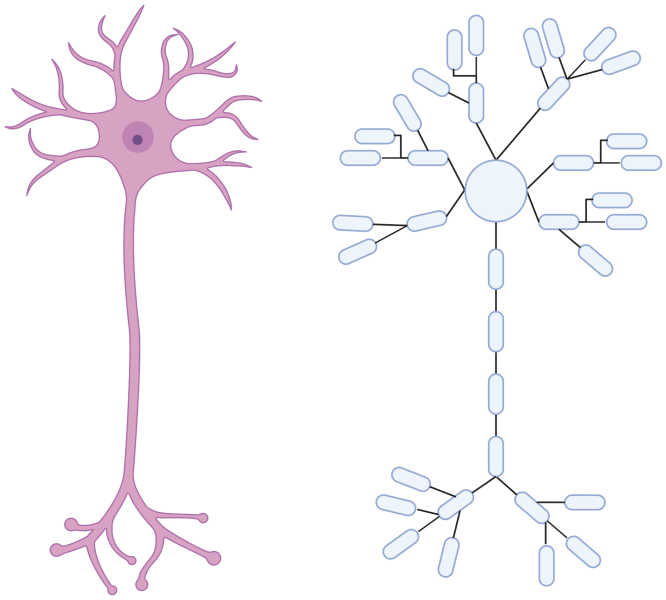

Figure 4: Drawing of neuron morphology on the left and the corresponding compartmental representation on the right. We can see that each branch of the neuron is represented by a series of compartments. It is possible to calculate a spectrum of values for these compartments; in this thesis, we will examine potential values in each compartment specifically and a spike initiation prediction. Image from Jonas Verhellen created with Biorender.com.

2.5 Compartmental Layer 5b Pyramidal Cell Model

This section will look at compartmental L5b PC models from "Models of Neocortical Layer 5b PCs Capturing a Wide Range of Dendritic and Perisomatic Active Properties" by Hay, Hill, Schürmann, Markman and Segev [6]. These models are based on neurons found in the mammalian cerebral cortex, specifically in L5b. The two main mechanisms explored by Hay et al. [6] are the dendritic and perisomatic active properties in L5b PCs. The dendritic active properties refer to the electrical dynamics in the neuron's dendrites, while the perisomatic active properties concern the electrical dynamics around the soma. Additionally, Hay et al. [6] looked at the interplay between the two zones and how they influence one another. The study by Hay et al. fitted a large set of experimental data from *in vitro* recordings of P36 Wistar rats;

they identified features of somatic and dendritic spike patterns and quantified their statistical properties [6]. The L5b cell models are conductance-based and use Hodgkin-Huxley formalism (see Section 2.3).

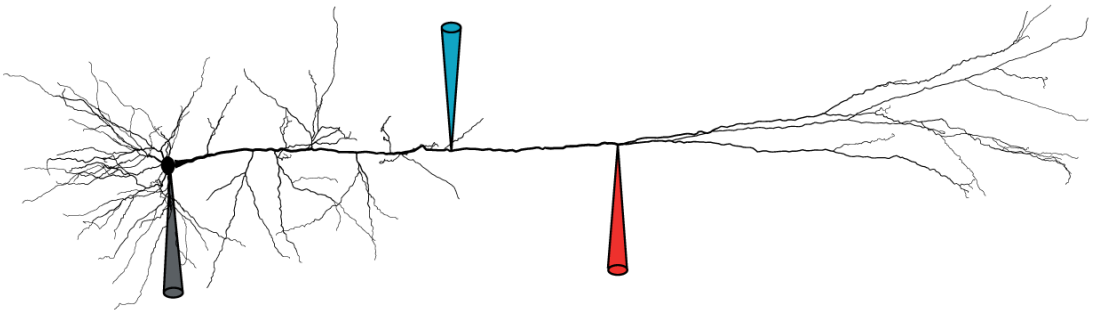

Figure 5: Morphology of an L5b neocortical PC. The study by Hay et al. [6] employs a combination of experimental data and computational modelling to capture the dynamics of the L5b PC. The black, blue and red electrodes represent the recording and stimulation sites used when collecting data. Reproduced from Figure 1 in Hay et al.[6].

L5b PCs are ideal for intracellular dendritic recordings due to the sizeable thickness of their dendrites. L5b PCs have therefore been studied thoroughly compared to many other neuron types. We have two primary spiking zones in L5b PCs; in the area surrounding the cell body, we have perisomatic Na^+ spikes, and at the distal apical dendrites [1] we have Ca^{2+} spikes. The two spiking zones have been shown to interact with each other by *in vitro* experiments, and it is, therefore, interesting to replicate both spiking zones in the same model [6].

2.5.1 BAC Firing and Perisomatic Step Current Firing

The models by Hay et al. [6] use an automated fitting method called Multi-Objective Optimisation (MOO) together with an evolutionary algorithm [22] when fitting models to experimental data. They fit models to 20 features gathered from experimental recordings, such as somatic and dendritic voltage responses. The authors emphasize two types of firing behaviours in their

[1] "Distal apical dendrites" refers to the farthest-reaching branches of the apical dendrites.

neural models; BAC[2] firing and perisomatic Na^+ step current firing. BAC firing occurs when an action potential initiated at the soma propagates into the dendrites, where calcium channels generate calcium spikes [23]. Perisomatic step current firing is essentially the firing pattern which arises when the neuron is injected with depolarizing and hyperpolarizing current injections at the soma; one of the earliest adoptions of current injections into the soma of a neuron was performed by Hodgkin and Huxley [4] (see Section 2.3).

Hay et al. aimed to develop acceptable models for both BAC firing and perisomatic Na^+ step current firing and incorporate the interplay between the two firing zones. They began by fitting the two firing types separately before trying to fit them jointly; first, they performed fitting of BAC firing targets, which led to some acceptable models; secondly, they tested these models on the other type of firing, the perisomatic step current firing before selecting the model which performed best for both target behaviours. This two-step method is how they optimised the features for both spiking zones [6]. The membrane capacitance was constant for the soma, axon and dendrites, while the specific membrane (leak) conductance was kept as a free parameter. They used Hodgkin- Huxley formalism when modelling the ionic currents (see Section 2.3):

$$I = \bar{g} \cdot p_m^{x_m} p_h^{y_h} \cdot (V - E_r) \tag{2.7}$$

where I is the ionic current, $\bar{g}$ is the maximal conductance, E_r is the reversal potential, V is the membrane potential for the ion involved, p_m is the activation variable, p_h is the inactivation variable, x_m is the number of activation variables and y_h is the number of inactivation variables. They used ten active ionic currents, incorporating kinetics from experimental literature.

[2]BAC: backpropagation action potential activated Ca^{2+} spike.

2.5.2 Hay et al. Study: Key Takeaways

Hay et al. [6] created plausible models that captured important characteristics observed in L5b PCs by optimising their models to reproduce both BAC firing and somatic step current firing properties. They used 20 experimentally-based features and their experimental variability to characterise the target behaviours of the two spiking zones. The proposed models from Hay et. al [6] capture the main characteristics of L5b PCs. We will be using data generated by the "Hay model" as our ground truth when we are training our deep learning models (see Section 3.2).

This thesis uses deep learning methods to study the input-output relationship from a model proposed by Hay et al. [6]. We want to learn the underlying mapping responsible for the transformation from input to output without having to solve the cable equations in the complex biophysical L5b PC models [24][25][26]. These cable equations are partial nonlinear differential equations used to model the compartmental potentials in spatially extended neuron models [12]. It can be challenging to understand and solve cable equations due to their spatial complexity [24] and the fact that they are composed of a high-order structure of coupled differential equations [8].

2.6 Local Field Potential (LFP)

In this thesis, we will employ neural networks to estimate voltage values within neural compartments; these compartmental voltage values enable the computation of transmembrane currents in a neuron, which can subsequently be utilized to estimate Local Field Potentials (LFPs). LFPs are low-frequency signals that reflect the interaction between neurons in the brain and are characterised by a frequency below 500 Hz [14] [27][28]. LFPs and action potentials are often measured together in electrophysiological experiments using micrometre-size electrodes [14] [29]. The local field potential measured at the electrodes is generated from transmembrane current passing through cellu-

lar membranes in the proximity of the electrode. The dominant component for LFP contribution seems to be from synaptic inputs since it is an essential source for extracellular current [30]. There is an established forward-modelling scheme based on multi-compartmental models to describe the contribution from neuron model activity to the extracellular potential $\phi(r_e,t)$ [28]:

$$\phi(r_e,t) = \frac{1}{4\pi\kappa}\sum_{n=1}^{N_c}\frac{I_n(t)}{|\mathbf{r_e}-\mathbf{r_n}|} \tag{2.8}$$

Where $I_n(t)$ is the transmembrane current in compartment n, $\mathbf{r_n}$ is the position of compartment n, $\mathbf{r_e}$ is the position of the tip of the electrode, N_c is the number of compartments in the model and κ is the extracellular conductivity. The variables $I_n(t)$, $\mathbf{r_n}$ and $\mathbf{r_e}$ are visualised in Figure 6. The dots inside each compartment in Figure 6 represent the assumption that the transmembrane currents enter the extracellular space from a single point based on the point source approximation. The superposition principle linearly combines the multiple current sources to acquire one local field potential [31]. The value of the conductivity κ will depend on how easily the transmembrane current can move through the extracellular medium, which is determined by separate experiments [28].

A few assumptions and approximations must be fulfilled for equation 2.8 to hold true. One of these is the quasistatic approximation of Maxwell's equations, where the electric (E) and magnetic (B) fields are essentially decoupled, which means that the terms $\partial E/\partial t$ and $\partial B/\partial t$ can effectively be ignored when calculating B and E, respectively [32]. We also have to assume an infinite volume conductor, and the electric conductivity κ has to be ohmic, isotropic, homogenous and frequency independent [28].

In Figure 7, we can see the distribution of extracellular potentials across a reconstructed L5b pyramidal neuron model. LFPs represent the summed activity of multiple neurons, which means we can represent larger brain sections by calculating LFPs. Furthermore, calculating LFPs and the corresponding dipole moments can be used to calculate measures such as EEG and

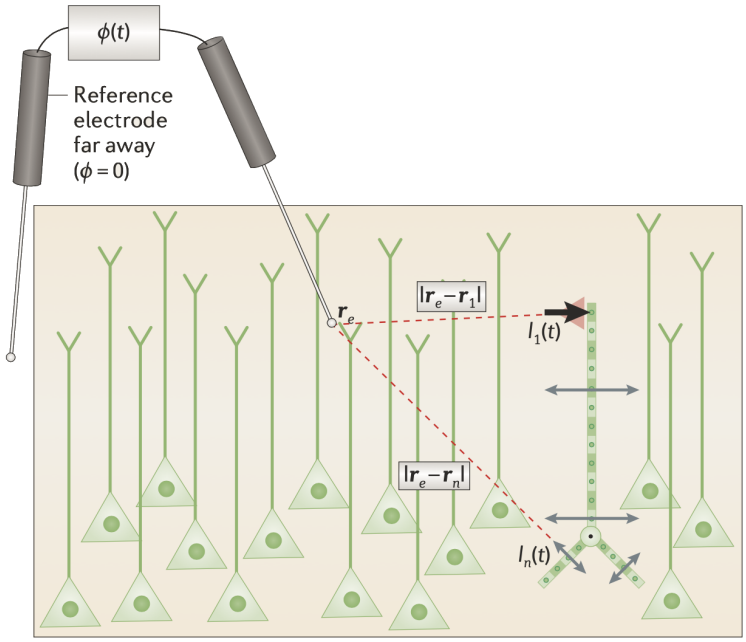

Figure 6: Depiction of how $I_n(t)$, $\mathbf{r_n}$ and $\mathbf{r_e}$ from equation 2.8 are positioned in relation to the neuron model and the electrode. The illustration depicts the single excitatory apical input as the red triangle onto a pyramidal neuron. The red dotted lines are the distance from the electrodes to the different compartments. The black arrow over the red triangle represents the transmembrane current from the synapse, while the grey arrows along the neuron represent the transmembrane return current [28]. Adapted from Einevoll et al. [28] with permission.

MEG signals [33], which can give us valuable insight into how these measures are related to each other. These simulated measures can then be compared with experimental LFP, EEG and MEG signals, and if the experimental data matches our simulations, it would indicate that our neuron model is accurately predicting some aspects of brain activity; it could therefore be advantageous to use our predicted compartmental voltage values from our DNNs to estimate LFPs.

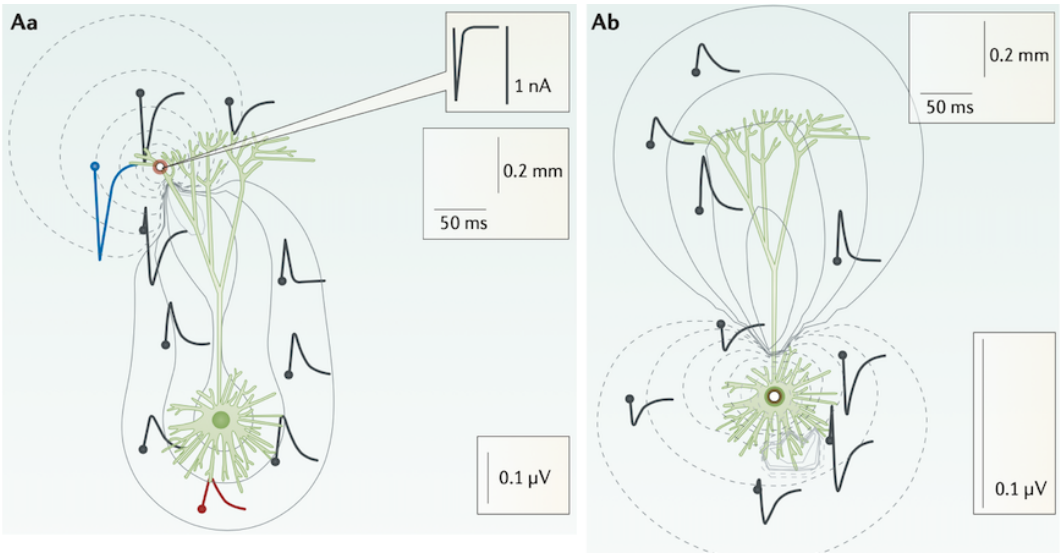

Figure 7: Aa and Ab are L5 PCs, where Aa is activated by a single excitatory synapse at one of the dendrites, and Ab is activated at the soma (the white circle indicates the activation position). The insert in the top corner in Aa is the shape of the injected current, the traces that start with a dot are the extracellular potentials in the different neuron compartments, and the gray contour lines are the LFP amplitudes. Reproduced from Einevoll et al. [28] with permission.

Chapter 3

Deep Artificial Neural Networks

In this chapter, we will explore the properties of our multi-task prediction methods and present a task-balancing method called Loss-Balanced Task Weighting LBTW. Additionally, we will introduce diversity metrics used to evaluate the diversity of our data representations. This chapter aims to describe the process of obtaining the underlying input-to-output mapping of the Hay model [6] by implementing deep learning MTL methods. Parts of Chapter 3 is based on chapter 3, 5, 6 and 9 in ”Deep Learning” by Goodfellow, Bengio and Courville [34].

3.1 Characterise Input and Output Mapping

We want to map the input/output relationship from the ”Hay model” introduced in Section 2.5. First, we will feed the input data used in the ”Hay model” [6] to our DNNs. Then, we will adjust the networks' weights according to the ”Hay model” output. The process will be similar to that attempted in the paper ”Single cortical neurons as deep artificial neural networks” by Beniaguev, Segev and London [8]. We will, however, incorporate a different architecture; the technical details of the architecture are explained in Section 3.7.

In practice, we will use MTL models to predict compartmental potential values and the outgoing spikes of an L5b neuron model, which will give us

insight into how the ionic currents behave in the neuron instead of only looking at the spiking behaviour as they did in Beniaguev et al. [8]. One of the primary advantages of employing neural networks over the conventional L5b model lies in the immense computational speedup that can be achieved by integrating a deep learning framework [8]. If the training of our neural networks are successful; we can generate a realistic overview of a section of a cortical area using both compartmental potentials and spikes.

3.2 Single Cortical Neurons as Deep Artificial Neural Networks

We will use an existing data set derived from the work of Beniaguev et al. [8] when training our MTL models. The data set is generated in NEURON and based on a L5b PC neuron model proposed in Hay et al. [6]. Beniaguev et al.[8] trained a DNN on this data to reflect the input-output behaviour of L5b cortical PCs by predicting the spiking properties of the neuron model. The input consisted of synaptic activity in 1278 synapses over 639 compartments, and the output consisted of outgoing spikes from the neuron. The DNN should ideally recreate the input-output transformation of the L5 PC model by changing the weights through a backpropagating learning algorithm (see Section 3.6). In Beniaguev et al. [8] they began by fitting a single-layer neural network to a Integrate and Fire (I&F) model as a proof of concept, where the learning process found one positive and one negative class of weights, corresponding to the excitatory and inhibitory synaptic inputs, respectively. The simple DNN learned the input-output transformation of an I&F model with a high degree of temporal accuracy [8].

Next, they applied the same framework to the L5 PC model [6], where the excitatory and inhibitory synapses were uniformly distributed across the dendritic tree. In this case, they had to increase the complexity of the neural network compared to the I&F model example. A Temporal Convolutional Neural Network (TCN) with seven 128-channel layers and a time window

of $T_w = 153$ ms was the first configuration that provided a satisfactory fit of the input-output relationship. This TCN managed to predict the somatic subthreshold voltage and the action potentials with high precision on unseen test data. The simulation speed for the neural network compared to the full L5 PC model was faster by several orders of magnitude, despite the considerable size of the neural network [8].

Beniaguev et al. [8] produced a DNN model that predicts the neurons spiking output at millisecond temporal resolution. They learned about the mechanics that shape the input-output function by analysing the weight matrices in their neural network. They found that the neural network generalised well for different stimulation patterns, where the size of the neural networks greatly influenced the generalisation capabilities; the deeper neural networks seemed to generalise better than networks with fewer layers [8]. We will train our networks on data from Beniaguev et al. [8]. In Beniaguev et al. [8], they only predicted the spiking output of the neuron model; we will attempt to predict both the voltages of the neuron compartments and spike generation. Our input data consists of synaptic input and the previous voltage values for each compartment. Our output data are the compartmental potential values and the outgoing spikes of the neuron.

3.3 Temporal Convolutional Neural Networks

The first section in our MTL models consists of the so-called ”experts” or ”shared bottom” in MH, which uses a generic TCN architecture, the temporal block. We will use a TCN architecture based on ”An Empirical Evaluation of Generic Convolutional and Recurrent Networks for Sequence Modeling” by Bai, Kolter and Koltun [35]. In neural networks, it is common to use general matrix multiplication for the hidden layers in a network. However, a convolutional neural network is characterised as using the convolution operation instead of matrix multiplication in one or more of the layers [34]. The convolution operation utilizes kernels, which can essentially be described as filters.

The output of the convolution operation is referred to as the feature map and is given by the integral over the multiplication of the input and kernel function:

$$(f*g)(t) = \int_{-\infty}^{\infty} f(a_d) * g(t-a_d) da_d \tag{3.1}$$

Where t is time, a_d represents the time delay (or age) of a measurement, $g(t-a_d)$ is the kernel/weighting function and $f(a_d)$ is the input signal at a time delay a_d [34]. When time is discretized, we can rewrite the general convolution operation as a sum:

$$(f*g)(t) = \sum_{-\infty}^{\infty} f(a_d) * g(t-a_d) \tag{3.2}$$

This is the general definition of the convolution operation but the limits of summation will of course change according to the dimensions of the input data. In our project, we are working with sequential input data consisting of binary synaptic inputs and 1-step delayed voltages at different time steps. In ordinary convolutional networks, it is common to perform the convolution on the elements surrounding both sides of the value we are looking at, which is not the case in TCNs, where we instead would like to use causal convolutions; when we perform our convolutional operations we only want to include elements of time t and earlier, not elements from the future [35], which is illustrated by the blue, red and green lines in Figure 8.

The output of the TCN must be the same length as the input, given that we want to represent the whole time sequence and keep the temporal structure of the data. We, therefore, add zero padding to the different hidden layers by the formula (kernel size - 1) * dilation size, where the kernel size is $k_s = 10$, and the dilation size is given by $d = 2^i$, $i = 0,1,2$; this will ensure that the filter is applied to each element in the hidden layers and it means that we can increase the dilation factor for each layer; see Figure 8 (in our final model we use a kernel size of $k_s = 10$ instead of $k_s = 2$). We can see that zero padding is added at the beginning of each layer to make sure that the receptive field

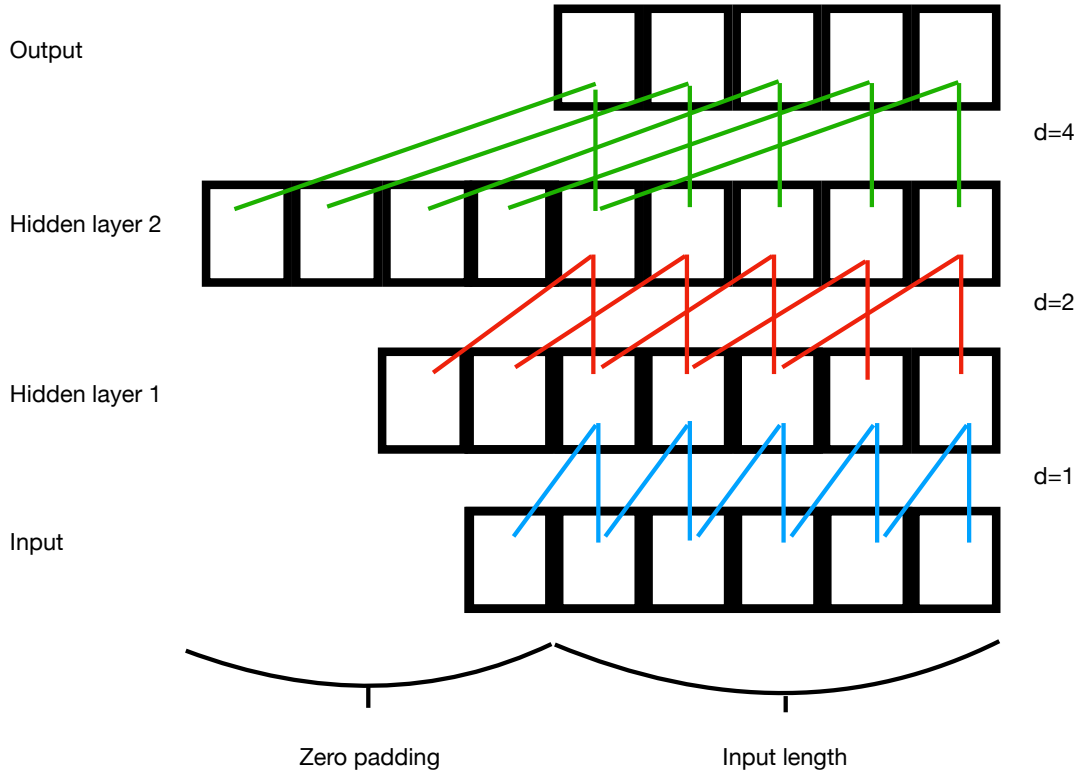

Figure 8: Depicted is a TCN structure with kernel size $k_s = 2$, meaning each element has two connections to the elements in the previous layer. The blue, red and green lines indicate the convolution operation between elements in adjacent layers. We have dilation factors $d = 1, 2, 4$. The dilation factor tells us how many elements we should skip from the previous layer when we perform our convolutions. Consider the following example: the element given by time step t in hidden layer 2 will skip the value from the previous time step $t-1$ and instead perform the convolution with t and $t-2$ in hidden layer 1, given that $d = 2$ between the layers. The "zero padding" length is given by $(k_s - 1) * d$, which gives a padding of 1, 2 and 4 for layers 1, 2 and 3, respectively. We can see that the causality condition is fulfilled, given that the output has the same length as the input. This depiction is a simplified drawing using a kernel size of $k_s = 2$ instead of $k_s = 10$ to represent the operations. Drawing by Sebastian Amundsen.

covers the whole input sequence to ensure causality [35]. By increasing the dilation factor between each layer, we perform dilated convolutions, which means we can extract valuable information from far-back past observations. We want to rewrite equation 3.2, where we include dilated convolutions and introduce the limits 0 to $k_s - 1$, where the input function f is $\mathbf{x}$, and we are looking at element s of the sequence:

$$(\mathbf{x} * g)(s) = \sum_{i=0}^{k_s-1} g(i) \cdot \mathbf{x}_{s-d \cdot i} \tag{3.3}$$

Where $g(i)$ is the kernel/filter, k_s is the kernel/filter size and $s - d \cdot i$ describes the direction of the past. By increasing the dilation factor d and the filter size k_s we can increase the receptive field, which means we can utilize a broader range of inputs [35]. The structure of the experts is described by a temporal block, where we use the following process: 1) Feed the input to a 1-dimensional convolutional network with weight normalization; 2) Add the padding; 3) Give the padded data to a sigmoid function; 4) Feed the data from the sigmoid function to the dropout function. These four steps are repeated in a sequential framework; see Figure 9.

We use the sigmoid activation function, which is commonly used in neural networks; see Appendix B for sigmoid equation and plot. We also use a method called dropout in our temporal block. The basic idea behind the dropout method is to remove certain nodes in our network layers given some probability, which could improve diversity in the network structure during training, and consequently contribute to less likelihood of overfitting [36]. The dropout function we are using will zero elements of the input tensor with a probability of $p_z = 0.2$. It is important to note that the dropout method is only active when training our network and not during network evaluation (testing). We have three temporal blocks in each expert (for MMoE/MMoEEx) and in the shared bottom (for MH).

The experts/shared bottom will ”pick up” important characteristics in the input data by ”sliding” learnable filters across the input. These filters can be

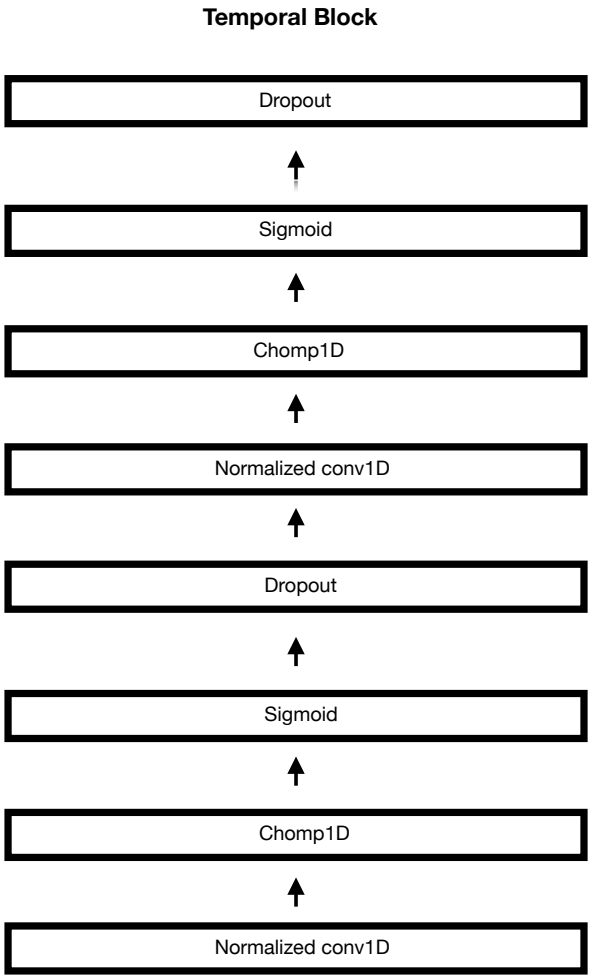

Figure 9: Illustration of the temporal block, where each section receives input from the previous section in sequential order. 1) The input data is first normalized and sent through a 1- dimensional convolution layer. 2) The convoluted data is sent to a chopping function which adds padding. 3) The sigmoid activation function is applied to the data. 4) The dropout function deactivates certain nodes in the network to improve generalization. These four steps are repeated one more time before receiving our output from the temporal block. We have three of these temporal blocks in the experts and the shared bottom. Drawing by Sebastian Amundsen.

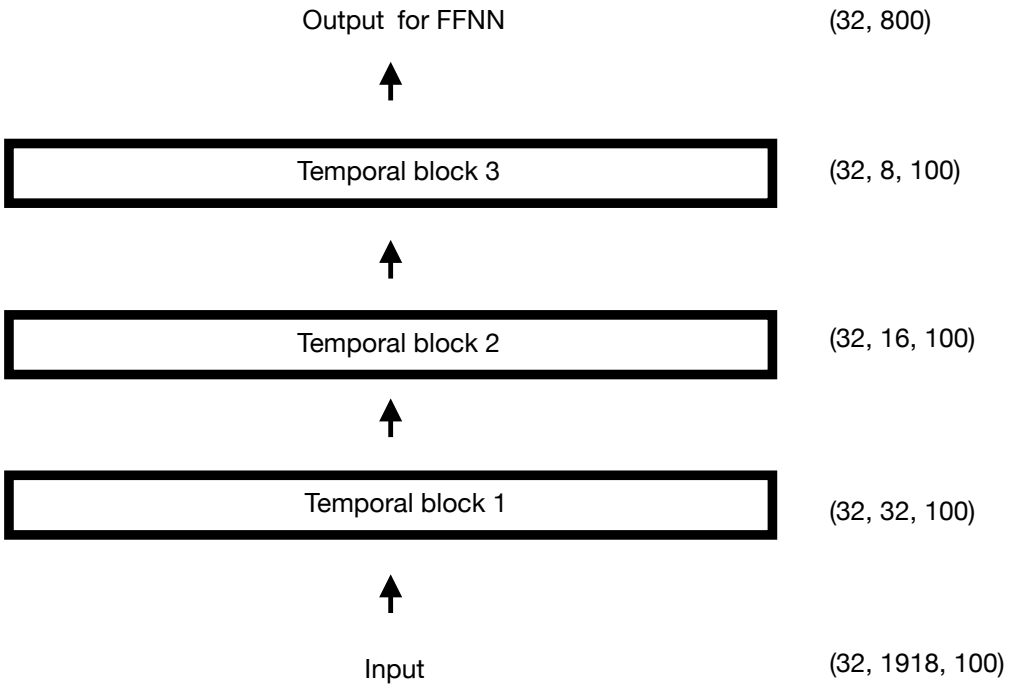

Figure 10: This figure depicts the three temporal blocks in the experts and the shared bottom. For the input, the dimensions are given by (Batch size, Features, Sequence length) and for the temporal blocks, the dimensions are (Batch size, Number of channels, Sequence length). The input with dimensions (32, 1918, 100) is sent to the first temporal block, where the 1918 features are represented as 32 channels. In blocks 2 and 3, these channels are reduced to 16 and 8, respectively. The final temporal block has dimensions (32, 8, 100). Before sending the data to our FFNNs/towers, we need to flatten the output, which gives dimensions (32, 800), where we have 32 batches with 800 elements/features. Drawing by Sebastian Amundsen.

seen as detectors, which look out for certain patterns. This pattern detection process generates representations of the input called channels, which represent feature maps of the data. The channels in our convolutional networks will decrease in size for each successive network. We have 32, 16 and 8 channels for the first, second and third temporal blocks. Each channel represents a feature map, which is adjusted by applying the kernel to the time sequence data. The dimensions of the components in the experts/shared bottom are given in Figure 10. In the MH model, the data configuration is sent from the shared bottom to feed-forward neural networks for further processing. By

having multiple experts, we can obtain different configurations of the input data, which can be beneficial for identifying features. In the MMoE/MMoEEx, the different data configurations are sent from the experts to feed-forward neural networks for further processing.

For deeper convolutional networks, it can be beneficial to incorporate residual connections that do not fit directly to the underlying mapping $\mathscr{H}(x)$ due to the degradation problem that occurs when deeper networks start to converge [37]. In "Deep Residual Learning for Image Recognition" by He, Zhang, Ren, and Sun [37], the authors propose a solution to this by introducing a "deep residual learning framework", where they fit a residual mapping $\mathscr{F}(x) := \mathscr{H}(x) - x$ instead of fitting directly to the desired underlying mapping $\mathscr{H}(x)$ [37]. Our TCN has few layers, so it is not necessary to implement such a framework. However, it may be beneficial to implement a residual mapping if one wants to increase the depth of the experts.

3.4 Feed Forward Neural Networks

The output from the experts/shared bottom is "fed" to feed-forward neural networks, which we refer to as towers. We have 641 towers in our deep learning models, each corresponding to a different task. Hopefully, the experts can find diverse enough data representations to specialize in different tasks and provide optimal input for each tower. For each task, we have a feed-forward neural network consisting of three linear layers with the exponential linear unit (ELU) activation function between each layer; see Appendix B for ELU function and visualisation.

We have different sizes for our towers in the soft-parameter models and hard-parameter models to make the sizes of the networks comparable; we, therefore, increase the number of units in the hard-parameter sharing model. For the hard-parameter model, the input layer has dimensions given by [TCN Output Size, Number of Units] = [800, 25]. The second layer has dimensions [Number of Units, Number of Units]= [25,25], and the third output layer

has dimensions [Number of Units, Task Length]= [25,1]. For the MMoE and MMoEEx model, we have 10 units instead of 25, which gives the architecture [800, 10] $\rightarrow$ [10, 10] $\rightarrow$ [10, 1] for the soft-parameter sharing models. The final output values from our towers are the predicted values we compare to the target data. Figure 11 shows the towers' structure. Our feed-forward neural network uses the ADAM optimiser and backpropagation to update the layers' weights, which will be discussed in Section 3.5 and 3.6, respectively.

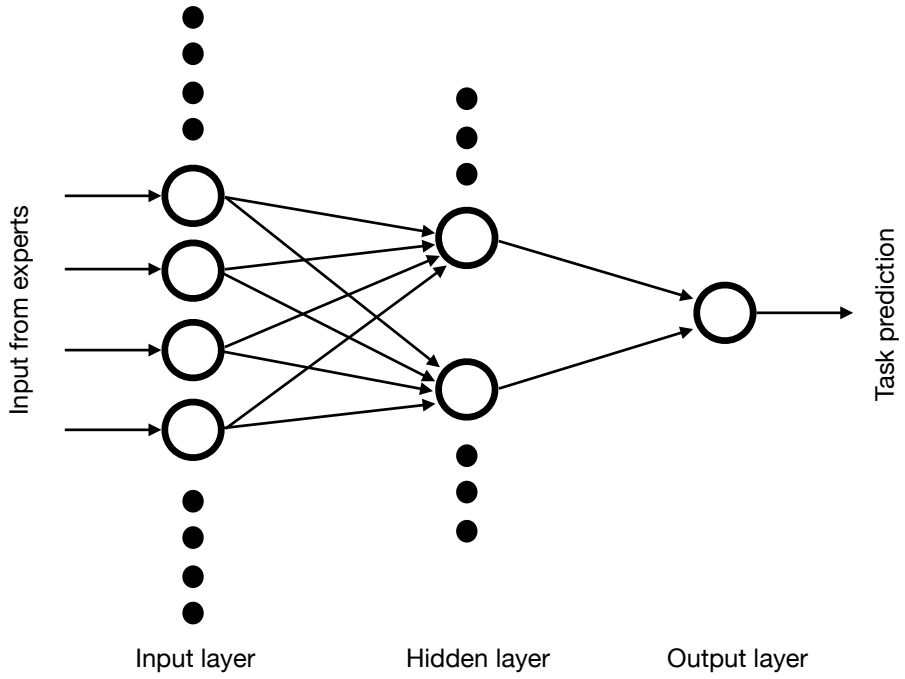

Figure 11: This is the feed-forward neural network structure we use for our towers. We only drew a section of the network to illustrate the key components of the full network. The dots represent the rest of the nodes in the tower. The input from the experts consists of 800 elements with a batch size of 32, which is given to the first input layer of 800 units. The hidden layer has 25 units for the hard-parameter model and 10 for both soft-parameter models. The output layer produces the prediction for the specific task of that tower. The arrows represent the different weights between each node. These weights will be adjusted by an optimiser and the backpropagation algorithm when we are training our network (see Section 3.5 and 3.6). Drawing by Sebastian Amundsen.

3.5 Loss Functions and Optimisers

The goal of our multi-task learning methods is to approximate transformation functions $\Xi_i(x)$ to target functions $\Xi_i^*(x)$, where $i = 0, 1, 2, 3, ..., 639, 640$. Our target functions represent the transformation between input and output data. As mentioned previously, the data consists of compartmental potential values and spikes gathered from the precomputed data set by Beniaguev et al. [8]. The difference between predicted and target values can be described using loss functions. We want our predicted output values to be as close as possible to the target values, which can be accomplished by minimizing the loss function. Multiple loss functions exist; we use the Mean Squared Error (MSE) function for the compartmental potential values and the Binary Cross Entropy with Logits Loss (BCELL) function for the spikes.

We want to minimize both the MSE and the BCELL in our deep learning models; we introduce backpropagation and the ADAM optimiser, which are methods used to update the weights in our neural network. Backpropagation is an algorithm which allows information from the loss function to propagate back through the network. We want to compute the gradient of the cost function $\nabla J(w,b)$ with respect to the parameters we are trying to optimise; these parameters are the weights w and the biases b.

3.5.1 Mean Squared Error (MSE)

We are using the MSE loss function when we compare our predicted potential values with the target potentials. The MSE is given by:

$$MSE(\hat{y}, y) = \frac{1}{n_b} \sum_{i}^{n_b} (\mathbb{A}(\hat{y}) - y)_i^2 \tag{3.4}$$

Where n_b is the batch size, $\mathbb{A}$ is the activation function, $\hat{y}$ is the prediction, and y is the target values. We apply the chain rule to 3.4 in order to obtain expressions for the derivative of MSE in relation to w and b:

$$\begin{aligned}\frac{\partial MSE}{\partial w} &= \frac{\partial MSE}{\partial \mathbb{A}}\frac{\partial \mathbb{A}}{\partial \hat{y}}\frac{\partial \hat{y}}{\partial w} \\ \frac{\partial MSE}{\partial b} &= \frac{\partial MSE}{\partial \mathbb{A}}\frac{\partial \mathbb{A}}{\partial \hat{y}}\frac{\partial \hat{y}}{\partial b}\end{aligned} \tag{3.5}$$

The gradient of the MSE loss function is given by:

$$\begin{aligned}\nabla MSE &= \left(\frac{\partial MSE}{\partial w}, \frac{\partial MSE}{\partial b}\right) = \left(\frac{\partial}{\partial w}\left(\frac{1}{n_b}\sum_i^{n_b}(\mathbb{A}(\hat{y})-y)_i^2\right), \frac{\partial}{\partial b}\left(\frac{1}{n_b}\sum_i^{n_b}(\mathbb{A}(\hat{y})-y)_i^2\right)\right) \\ &= \left(\frac{2}{n_b}\sum_i^{n_b}(\mathbb{A}(\hat{y})_i - y_i)\frac{\partial \mathbb{A}}{\partial \hat{y}}\frac{\partial \hat{y}}{\partial w}, \frac{2}{n_b}\sum_i^{n_b}(\mathbb{A}(\hat{y})_i - y_i)\frac{\partial \mathbb{A}}{\partial \hat{y}}\frac{\partial \hat{y}}{\partial b}\right)\end{aligned} \tag{3.6}$$

The general expressions $\frac{\partial \mathbb{A}}{\partial \hat{y}}\frac{\partial \hat{y}}{\partial w}$ and $\frac{\partial \mathbb{A}}{\partial \hat{y}}\frac{\partial \hat{y}}{\partial b}$ will be calculated when we propagate back through our network using the backpropagation algorithm (see Section 3.6).

3.5.2 Binary Cross Entropy with Logit Loss (BCELL)

The BCELL function combines the sigmoid function (see Appendix B) with the binary cross entropy loss. The sigmoid function's purpose is to transform the output from the towers to probabilities. These probabilities are then compared to the binary probability of generating a spike using the cross entropy loss. The BCELL loss function is given by:

$$\text{BCELL}(\hat{y}, y) = \frac{1}{n_b}\sum_i^{n_b} -\Big(y_i \cdot \log\big(\mathbb{A}(\hat{y}_i)\big) + (1-y_i)\cdot \log\big(1-\mathbb{A}(\hat{y}_i)\big)\Big) \tag{3.7}$$

We also want an expression for the gradient of the BCELL loss function:

$$\begin{aligned}\nabla \text{BCELL}(\hat{y}, y) &= \left(\frac{\partial l(\hat{y}, y)}{\partial w}, \frac{\partial l(\hat{y}, y)}{\partial b} \right) \\ &= \left(\frac{\partial}{\partial w} \left(\frac{1}{n_b} \sum_{i}^{n_b} -\left(y_i \cdot \log\left(\mathbb{A}(\hat{y}_i) \right) + (1 - y_i) \cdot \log(1 - \mathbb{A}(\hat{y}_i)) \right) \right) \right. \\ &\left. , \frac{\partial}{\partial b} \left(\frac{1}{n_b} \sum_{i}^{n_b} -\left(y_i \cdot \log\left(\mathbb{A}(\hat{y}_i) \right) + (1 - y_i) \cdot \log(1 - \mathbb{A}(\hat{y}_i)) \right) \right) \right)\end{aligned} \tag{3.8}$$

Where $\mathbb{A}$ is the activation function. We need to apply the chain rule to 3.8:

$$\begin{aligned}\frac{\partial \text{BCELL}}{\partial w} &= \frac{\partial \text{BCELL}}{\partial \mathbb{A}} \frac{\partial \mathbb{A}}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial w} \\ &= -\left(\frac{y}{\mathbb{A}(\hat{y})} - \frac{(1 - y)}{1 - \mathbb{A}(\hat{y})} \right) \frac{\partial \mathbb{A}}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial w}\end{aligned} \tag{3.9}$$

We calculate the expression for the derivative with respect to the bias b in the same way as above. We can now write a general expression for the gradient of BCELL:

$$\nabla \text{BCELL}(\hat{y}, y) = \left(-\left(\frac{y}{\mathbb{A}(\hat{y})} - \frac{(1 - y)}{1 - \mathbb{A}(\hat{y})} \right) \frac{\partial \mathbb{A}}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial w}, -\left(\frac{y}{\mathbb{A}(\hat{y})} - \frac{(1 - y)}{1 - \mathbb{A}(\hat{y})} \right) \frac{\partial \mathbb{A}}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial b} \right) \tag{3.10}$$

As for the MSE function, the general expressions $\frac{\partial \mathbb{A}}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial w}$ and $\frac{\partial \mathbb{A}}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial b}$ will be calculated when we propagate backwards through our network using the backpropagation algorithm (see Section 3.6).

3.5.3 ADAM Optimiser

It is important to choose a suitable optimisation algorithm when we are learning the weights of our network. We use optimisation algorithms to find the optimal values to minimize or maximize a function. In our case, we want to minimize the loss function(s). We will be using an adaptive learning rate optimisation algorithm called the Adaptive Moment Estimation (ADAM) algorithm, which is introduced in "ADAM: A Method For Stochastic Optimiza-

tion" by Kingma and Ba [38]. There have to be set some requirements before executing the ADAM algorithm; we set the learning rate $\eta = 0.001$, the exponential decay rates $\rho_1 = 0.9$ and $\rho_2 = 0.999$, the stabilisation factor $\nu = 1 \times 10^{-8}$, the weight decay $\Lambda = 0.0001$, the time step $dt = 1$ and define the initial weights w_i and biases b_i. The ADAM algorithm is detailed in **Algorithm 1**.

Algorithm 1 ADAM Optimisation Algorithm [38][34]

Initialize time t = 0

Initialize moment variables $q = 0$ and $r = 0$

Initialize parameters w_i and b_i

Define inital paramters $\theta(w,b)$

while criteria is met **do**

Take minibatch sample of size n_b from training set $\{x^!, x^2, ..., x^{n_b}\}$

with targets $\{y^!, y^2, ..., y^{n_b}\}$

Compute the gradient: $G = 1/n_b \sum_i^{n_b} \nabla_\theta J(\theta)$

$t = t + dt$

$G = G + \Lambda\theta_{t-dt}$

$q = \rho_1 q + (1-\rho_1)G$

$r = \rho_2 r + (1-\rho_2)G^2$

Correct bias: $\hat{q} = q/(1-\rho_1^t)$

Correct bias: $\hat{r} = r/(1-\rho_2^t)$

Compute the update: $\Delta\theta = -\eta\hat{q}/(\sqrt{\hat{r}} + \nu)$

Update the parameters: $\theta = \theta + \Delta\theta$

end while

Here we have introduced the moment variable q (the mean) and r (the uncentered variance) in conjunction with two exponential decay rates for the moment estimates ρ_1 and ρ_2. These moment estimates of the mean and the variance are initialized at zero, which means they are biased towards zero, particularly at the initial time steps and when the weight decay Λ is small (in our case). It is, therefore, beneficial to correct the biases in the momentum with the terms ρ_1^t and ρ_2^t to obtain $\hat{q}$ and $\hat{r}$, where t refers to the iteration

number [38].

The ADAM optimiser combines the AdaGrad [39] and the RMSProp [40] method. The AdaGrad method works well with sparse gradients, which are gradients where many components in the gradient vector are zero (or close to zero). The RMSProp method works well for incremental learning and when the underlying data distribution changes over time [38] (see [41] for incremental learning). The ADAM optimiser combines the benefits of both of these methods. The main differences between the ADAM and the RMSProp are: 1) ADAM updates its parameters using a running average of the first q and second r moment, while the RMSProp updates its parameters using momentum on the rescaled gradient; 2) RMSProp does not have a bias- correction term [38], while ADAM does. The main difference between AdaGrad and ADAM concerns how they adjust the learning rate for each model parameter during training: ADAM uses q and r to update the learning rate for each parameter, while AdaGrad takes the square root of the sum of past gradients to adapt the learning rate of the model parameters [39].

3.6 The Backpropagation Learning Algorithm

We have described how the weights and biases are updated in our deep learning models; the remaining part is to explain how to propagate back through the network to update these parameters. This propagation method is referred to as the backpropagation learning algorithm (backprop), and it is utilized in both the experts and towers. We can see that the gradient of the cost functions 3.6 and 3.10 are not fully written out; this is because we need to calculate the derivation term(s) in relation to the weights and biases for each node in every layer that is propagated backwards to. The backpropagation calculus in this section is based on chapter 2 in ”Neural Networks and Deep Learning” by Nielsen [42]. This section will propose a general expression for the backpropagation algorithm.

3.6.1 Notation

We will introduce a neat notation to describe the general idea behind back-prop, where the feed-forward neural network structure will be the reference point for our explanations. We define our weights w^l_{jk} as the lines from layer $l-1$ to layer l, where j specifies the node in layer l and k is the node in layer $l-1$. The biases b^l_j and activations a^l_j are defined similarly, where each node has a bias b^l_j and activation a^l_j. In our case, the activation function is the ELU for our towers, and the Sigmoid (see Appendix B) for our experts, but the general expression for a^l_j given an arbitrary activation function $\mathbb{A}$ is:

$$a^l_j = \mathbb{A}\left(\sum_k^K w^l_{jk} a^{l-1}_j + b^l_j\right) \tag{3.11}$$

Where K is the number of nodes in layer l. In Figure 12, we can see how the weight from node 6 in layer 3 is connected to node 1 in layer 3. The figure also depicts the activation at node 1 in layer 3 and the bias at node 5 in layer 2. We can rewrite equation 3.11 in a vectorized form:

$$a^l_j = \mathbb{A}(w^l a^{l-1} + b^l) \tag{3.12}$$

Where there is a weight matrix $w^l \in \mathbb{R}^{K\times M}$ for the specific layer l, with K defined as the number of nodes in layer l and M defined as the number of nodes in layer $l-1$:

$$w^l = \begin{bmatrix} w^l_{11} & w^l_{12} & \cdots & w_{1M} \\ w^l_{21} & w_{22} & \cdots & w_{2M} \\ \vdots & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \vdots \\ w_{K1} & w_{K2} & \cdots & w_{KM} \end{bmatrix} \tag{3.13}$$

This matrix gives that the entry in row j and column k is w^l_{jk}. Note that the dimensions of the matrix $K \times M$ will change according to how many nodes there are in each layer. In 3.12 there is also used vector representations for

the bias b^l and the activation a^l:

$$\begin{aligned} b^l &= (b_1^l, b_2^l, b_3^l, ..., b_K^l) \\ a^{l-1} &= (a_1^{l-1}, a_2^{l-1}, a_3^{l-1}, ..., a_M^{l-1}) \end{aligned} \tag{3.14}$$

Where the elements $(1,2,3,.....K)$ in the bias vector b^l are the nodes in layer l and the elements $(1,2,3,.....M)$ in the activation vector a^l are the nodes in layer $l-1$. We will introduce an abbreviation for the term inside the activation function: $z^l = w^l a^{l-1} + b^l$. There is also an error term ε_j^l, which represents the errors corresponding to layer l:

$$\varepsilon_j^l \equiv \frac{\partial J}{\partial z_j^l} \tag{3.15}$$

3.6.2 Method

We want to compute the partial derivatives $\partial J/\partial w_{jk}^l$ and $\partial J/\partial b_j^l$ for each layer l in our network. To do this, we first have to introduce four key equations. We begin by looking at the error vector ε^L that contains the errors for each node j in the final output layer L:

$$\varepsilon^L = \frac{\partial J}{\partial a^L} \odot \frac{\partial a^L}{\partial z^L} \tag{K1}$$

Where the Hadamard product $\odot$ is used for the vector product. The error vector ε^L essentially becomes our initial condition, and we must go back through the layers to compute the errors for the remaining layers. The general expression for the error vector ε^l in layer l is given by the error vector ε^{l+1} in the upcoming layer $l+1$:

$$\varepsilon^l = \left((w^{l+1})^T \varepsilon^{l+1} \odot \frac{\partial a^L}{\partial z^l} \right) \tag{K2}$$

The weight matrix w^{l+1} determines how the activations of the nodes in

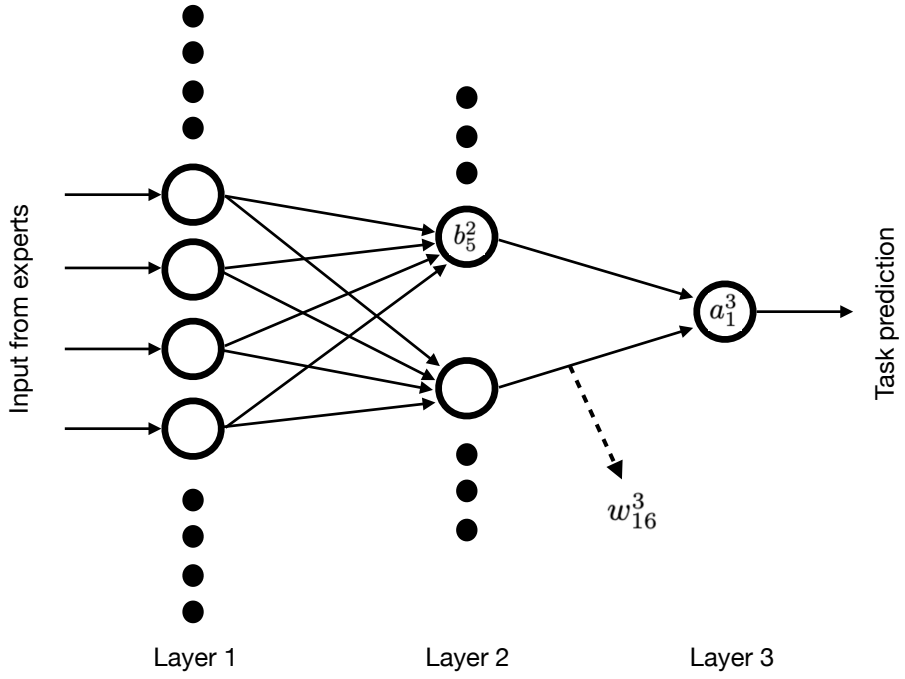

Figure 12: Depiction of how the weight w_{jk}^l, activation a_j^l and bias b_j^l are positioned according to our indexing scheme. The weight is depicted as the connection between node 6 in layer 2 and node 1 in layer 3. The activation at node 1 in layer 3 and the bias at node 5 in layer 2 are also shown. It may be confusing that the nodes in the second layer are indexed as 5 and 6; however, the reasoning is based on the rationale that these nodes are positioned in the middle of the second layer. Drawing by Sebastian Amundsen.

layer l contribute to the activations of the nodes in layer $l+1$. The transpose of the weights $(w^{l+1})^T$ will essentially reverse the direction of the connections so that we can propagate the error backwards by multiplying it with ε^{l+1}. In equation K2 the Hadamard product is applied to the $\frac{\partial a^L}{\partial z^l}$ term which essentially propagates the error term $(w^{l+1})^T \varepsilon^{l+1}$ in reverse through the activation function $\mathbb{A}$ in layer l. We can now compute the error ε^l for each layer l by using equation K1 as the initial condition for layer L and then repeat equation K2 for each consecutive layer $(L-1, L-2, L-3, ..., 2)$ in the network. It is important to note that both ε^l and ε^L are vectors consisting of errors for all nodes K in layer l and L, respectively, which means that we can access the specific error at node j with the notation ε_j^l. The derivative of the cost function $J(w,b)$

can now be expressed with respect to the bias b_j^l at node j in layer l:

$$\frac{\partial J}{\partial b_j^l} = \varepsilon_j^l \tag{K3}$$

Finally, we have an equation for the derivative of the cost function $J(w,b)$ with respect to the weights w_{jk}^l:

$$\frac{\partial J}{\partial w_{jk}^l} = a_k^{l-1}\varepsilon_j^l \tag{K4}$$

These four fundamental equations are everything needed to describe the backprop process. We can now find the change in our cost function $J(w,b)$ as we adjust the weights and biases of the network. We begin by applying equation K1 to the last layer in our neural network, which will give us the error in the output layer L. We then recursively apply equation K2 for each prior layer l, all the way down to the first layer in the network. We can then use equation K3 and K4 to calculate the derivative of the cost function $J(w,b)$ with respect to the weights w_{jk}^l and biases b_j^l for each node j in every layer l, before applying the ADAM algorithm (**Algorithm 1**) to update the weights and biases for every node in the network, given that we now know the gradient of the cost function for every node in the network. To summarize, we can apply these four fundamental equations together with the optimiser to update the weights and biases of our network based on the minimization of both the MSE (eq 3.4 for potential prediction) and the BCELL (eq 3.7 for spike prediction) loss function.

3.7 Multi-Expert Multi-Prediction Neural Networks

One of the most common approaches in machine learning is the Single-Task Learning (STL) archetype, where models predict tasks independently. STLs can generate satisfactory results when predicting independent tasks. However, STLs may be insufficient when predicting correlated tasks with shared

features; if two output targets are associated with the same input data, we still have to train two separate independent models for each task [11]. We can save resources and time if we share the underlying features across tasks.

In contrast to the STLs, we have the MTL models, which optimise a single model to perform multiple related tasks simultaneously, hopefully improving efficiency. Furthermore, MTL models appear ideal for our specific prediction problem, given that we predict compartmental potentials dependent on the same synaptic input (each compartment is stimulated separately at the same time step t). Therefore, we will be looking at three different MTL models with a varying degree of complexity; 1) The Multi-task Hard-parameter sharing (MH) model proposed in "Multitask Learning: A Knowledge-Based Source of Inductive Bias" by Caruana [9]; 2) The Multi-gate Mixture-of-Experts (MMoE) model proposed in "Modeling Task Relationships in Multi-task Learning with Multi-gate Mixture-of-Experts" by Ma, Zhao, Yi, Chen, Hong and Chi [10]; 3) The Multi-gate Mixture-of-Experts with Exclusivity (MMoEEx) model proposed in "Heterogeneous Multi-task Learning with Expert Diversity" by Aoki, Tung and Oliveira [11].

We can divide the MTL models into hard-parameter and soft-parameter sharing models. The MH method is an example of a hard-parameter sharing model, where the bottom layer of the neural network is shared between all tasks, and the top layers are task-specific [11]. The soft-parameter sharing models incorporate multiple experts in their bottom layer, which are shared across tasks by a feature-sharing mechanism that allows tasks to train on different feature representations. Both MMoE and MMoEEx are examples of soft-parameter sharing methods. The sharing mechanism in these models is called the gate function, which selects a subset of experts for each task, subsequently enabling feature sharing [11]. The top layers of the soft-parameter models are task specific.

3.7.1 MH

The Multi-task Hard-parameter sharing (MH) model is the most fundamental of the three MTL models. The first section of the model consists of three temporal blocks, which we refer to as the shared bottom. The temporal blocks process the raw data (see Section 3.3), generating a feature representation. The data is then sent to task-specific towers (see Section 3.4). The final output will be either a voltage prediction or the spike prediction. Next, these predicted values $\hat{y}$ are compared with the target values y by producing a loss value. The architecture is illustrated in Figure 13. After the data has passed through the architecture, we use the ADAM optimiser and backpropagation (see Section 3.5 and 3.6) to update the weights and biases for the feed-forward neural networks and the shared bottom.

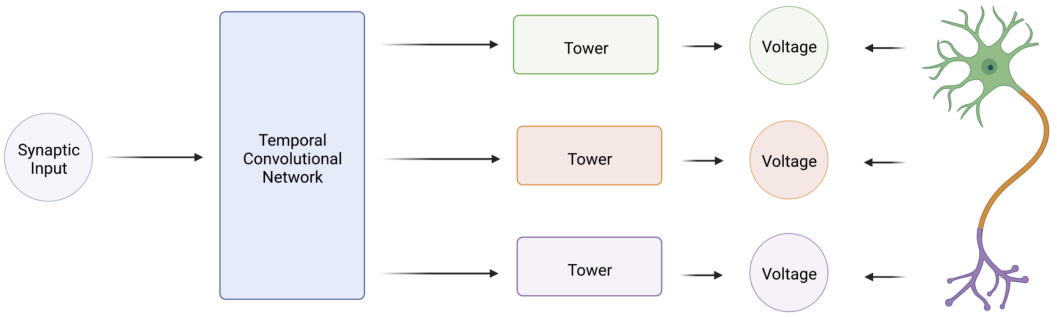

Figure 13: This is the architecture of the MH model. The shared bottom consists of three temporal blocks, which generate a feature representation of the data. The processed data is then sent to task-specific towers consisting of feed-forward neural networks. The final output will be either a voltage prediction or the spike prediction. Next, these predicted values are compared with the target values. Finally, we use backpropagation and the ADAM optimiser to calculate the gradients of the loss function and adjust the weights and biases in the shared bottom and the towers. Image from Jonas Verhellen created with Biorender.com.

We refer to the shared bottom layer as a function $\hat{s}$, and the task-specific

towers are defined as h^{τ}, where $\tau = [0, 1, 2, 3, 4, ..., T]$, where T is the number of tasks. The output $\hat{y}_{\tau}$ from each tower can be expressed as:

$$\hat{y}_{\tau} = h^{\tau}(\hat{s}(x)) \tag{3.16}$$

Where x is the input [10]. The final output of the MH model $\hat{y}_{\tau}$ is the voltages for each of the 640 compartments, in addition to a spike occurrence prediction. One of the main advantages of the MH model is its scale invariance to an increasing number of tasks, as the complexity of the shared bottom does not change; it is only the number of task-specific towers that change in accordance with the number of tasks being addressed. However, the method can be susceptible to specialising in the tasks with the strongest signal and consequently provide insufficient generalisation [11].

3.7.2 MMoE

The first soft-parameter MTL method we will look at is the Multi-Gate Mixture of Experts (MMoE) model. In this model, we have multiple experts combined with gates instead of only having one shared bottom, as we did in MH. The experts will learn different input data representations, hopefully leading to greater generalization than we had in the hard-parameter model (MH). The neural input is first sent to the gates of the MMoE network. The gates determine the importance of each expert in relation to the given input data. The data is then sent to the different experts in our network, where different data configurations are generated (see Section 3.3).

Afterwards, we multiply the weights of the gates and the weights of the experts to determine the experts that should have the highest contribution to the combined output from the experts to one specific task. Finding the optimal expert combination is done for every task we want to predict. The features from the experts are then sent to task-specific towers that process the data and make either a voltage value prediction or the spike initiation prediction (see Section 3.4). The tower layout of the MMoE is identical to that

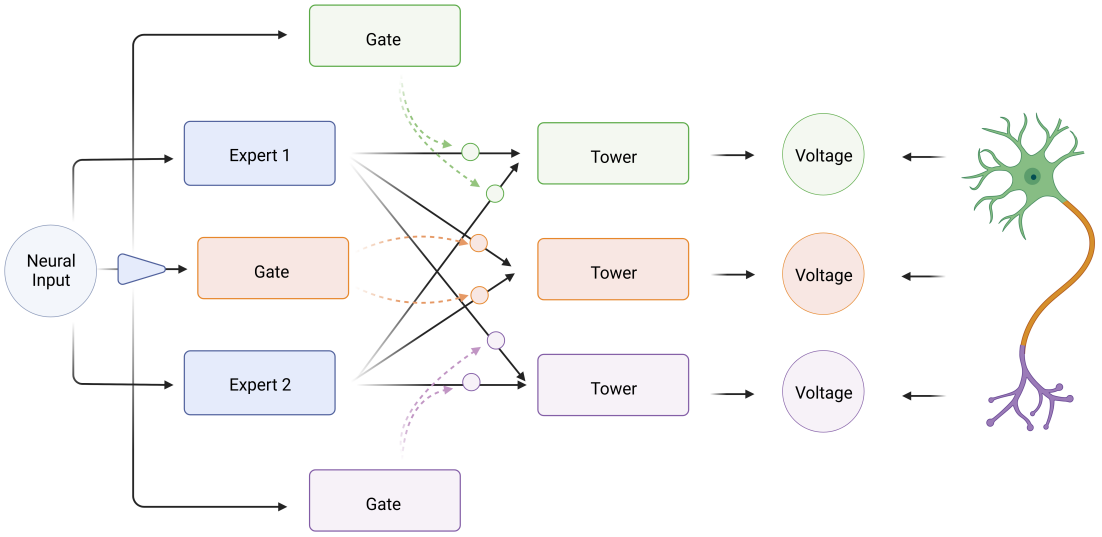

Figure 14: Here we have an illustration of the MMoE model. The first section of the MMoE consists of gates and experts. The neural input is sent to the gates, which determine the importance of each expert concerning the neural input data for every task. Afterwards, the data is processed by the temporal blocks in the experts, and different feature representations are generated. These feature representations of the time sequence data are combined with the weights in the gates and sent to task-specific towers, and the final output is produced. The final output will be voltage predictions for the compartments and a spike prediction. We compare the predicted values with the target values using a cost function. We then want to minimize the cost function $J(w,b)$ by adjusting the weights and biases of the network with the help of backpropagation and the ADAM optimiser. Image from Jonas Verhellen created with Biorender.com.

in the MH model, where each task has a separate feed-forward neural network. The predictions $\hat{y}$ are compared with the target values y using the MSE loss function for the compartmental voltages and the BCELL loss function for the spike prediction. After the data has passed through the architecture, we use the ADAM optimiser and backpropagation (see Section 3.5 and 3.6) to update the weights and biases for the gates, experts and towers. The MMoE architecture is depicted in Figure 14.

In MMoE, the experts are combined using gate functions. The gate function $\hat{g}^{\tau}$ for each task τ is given by:

$$\hat{g}^{\tau}(x) = S(M_{g\tau}x) \tag{3.17}$$

Where S is the softmax function and $M_{g\tau} \in \mathbb{R}^{n_e \times d_f}$ is a trainable matrix with n_e experts and feature dimension d_f. The softmax function $S(x)$ is given by:

$$S(x)_i = \frac{e^{x_i}}{\sum_j^T e^{x_j}} \tag{3.18}$$

Where x is the input tensor. The term in the denominator is the normalization term, which ensures that the function's output values sum to 1 and that all values are in the range (0,1). The softmax function can be interpreted as an extension of the sigmoid function (see Appendix B), where sigmoid is used for binary classification while softmax applies to multiple classes T. The combination of gates and experts is given by:

$$f^{\tau}(x) = \sum_{i=1}^{n_e} \hat{g}^{\tau}(x)_i \hat{s}_i(x) \tag{3.19}$$

Where we sum over the number of experts n_e. The final output for a task τ is given by:

$$\hat{y}_{\tau} = h^{\tau}(f^{\tau}(x)) \tag{3.20}$$

In practice, the gating networks can learn to interact with a select subset of experts, and the MMoE can figure out which gates overlap with the same experts. If tasks that share the same experts are not related, the gating networks of these tasks will attempt to use different experts [10].

3.7.3 MMoEEx

The other soft-parameter model we will examine is the Multi-gate Mixture-of-Experts with Exclusivity (MMoEEx) model. The architecture is almost identical to the MMoE model; the only difference is that MMoEEx incorporates a method to increase expert diversity. In MMoE, the diversity among the experts only stems from the random initialization of the experts and the

assumption that the gates will provide sufficient diversity; this means that there is no guarantee that the experts will be diverse enough to specialize in the different tasks; the MMoEEx attempts to address this diversity concern. In [11], they propose two properties to increase diversity among experts; exclusivity and exclusion.

The exclusivity mechanism sets a portion of experts αn_e to be exclusively connected to one task, where n_e is the number of experts. The exclusivity term α is defined in the range $\alpha \in (0, 1)$. Higher values of α mean more experts are exclusive, while lower values mean fewer exclusive experts; $\alpha = 1.0$ would make all experts exclusive and $\alpha = 0$ would mean all experts are shared. Exclusive experts are randomly assigned to tasks, however; one of these tasks can still be associated with other shared and exclusive experts. In practice, we would probably make a handful of the experts exclusive while the rest remain shared; this is at least true for our case where we set $\alpha = 0.1$, which means that 10 % of the experts are exclusive and the rest are shared between all tasks. The exclusion mechanism randomly excludes connections between experts and tasks. Larger α values mean more connections are removed, while lower α values mean fewer connections are removed; $\alpha = 1.0$ would make every expert lose one connection, and $\alpha = 0$ means that no connections would be removed.

Exclusivity and exclusion are mechanisms that close some gates for specific experts, which can increase diversity among experts. Other than the exclusivity and exclusion properties, the rest of the MMoEEx architecture is identical to the MMoE model; see Figure 15. Ideally, the diverse mechanisms of the MMoEEx should boost the generalisation performance compared to the traditional MMoE. It is important to note that the MMoE and the MMoEEx have some scalability issues due to the fact that the size of the MTL network tends to grow proportionally with the number of tasks [11]. One key difference in implementation for our models and the one proposed in Aoki at al. [11] is that they used LSTMs for the experts while we used temporal blocks. Furthermore, it is common to incorporate task balancing in MTL methods to

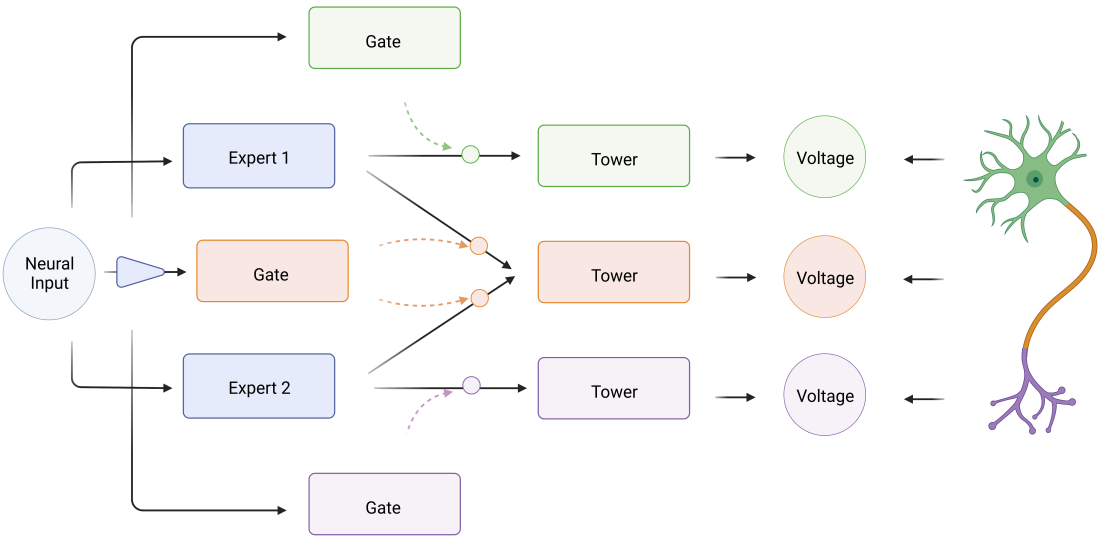

Figure 15: Here we have an illustration of the Multi-gate Mixture of Experts with Exclusivity (MMoEEx) model. The first section of the MMoEEx consists of gates and experts. The neural input is sent to the gates, which determine the importance of each expert in relation to the neural input data for every task. MMoEEx also introduces an exclusivity mechanism where some experts are ”closed off” for certain tasks. The temporal blocks in the experts process the neural input data and generate different feature representations. These feature representations of the time sequence data are combined with the weights in the gates and sent to task-specific towers, and the final output is produced. The final output will be voltage predictions for the compartments and a spike prediction. We compare the predicted values with the target values using a cost function. We then want to minimize the cost function by adjusting the weights and biases of the network with the help of the ADAM optimiser and backpropagation. Image from Jonas Verhellen created with Biorender.com.

reduce the possibility of certain tasks dominating the optimisation [11]. We will attempt to implement one of these task-balancing methods and compare the results with the ordinary MTL methods (see Section 3.8).

3.8 Loss-Balanced Task Weighting - LBTW

The main goal of task-balancing methods is to deal with negative transfer. Negative transfer is characterised as a reduction in performance for one or multiple tasks by the diversification of the network. In practice, for some tasks, it could be better to utilize methods which do not use feature sharing across tasks. To prevent negative transfer, we incorporate a method to balance the task gradients and prevent any of them from dominating the network. There are several different optimisation techniques we can implement to reduce negative transfer; in [11], they proposed a Loss Balance Task Weighting (LBTW) method [43], which is the method that we will implement in our MTL models. The LBTW method assesses the loss ratio between the current and initial loss for each task and then dynamically changes the weights of the different task losses to regulate task priority. The LBTW algorithm is detailed in **Algorithm 2** given that we have T tasks and a balance parameter α_b.

It is important to note that we need to perform an extra step for the voltage losses $L_{\hat{y}_{\text{voltages}}}$ before we can add it to the total loss. We need to take the mean of the MSE for all the batches manually before we multiply the total $\hat{\text{MSE}}$ loss with the weights since the MSE function is squared (see equation 3.4). Afterwards, we divide the loss by the length of the weights to normalize the overall $\hat{\text{MSE}}$ loss, and then we multiply the loss by the weights:

$$L = \frac{\hat{\text{MSE}}(\hat{y}_{\text{voltages}}, y_{\text{voltages}}) \cdot \lambda_\tau}{||\lambda_\tau||} + \text{BCELL}(\hat{y}_{\text{spikes}}, y_{\text{spikes}}) \cdot \lambda_T \tag{3.21}$$

Since the BCELL loss function has no squared terms, we can simply mul-

Algorithm 2 Loss-Balanced Task Weighting (LBTW)

Given T task and parameter α_b
Set the initial losses $L_{\hat{y}}$ to zero.
Set initial task weights λ_τ to one.
for each epoch i **do**
  for each batch in voltage predictions **do**
    Calculate loss for spike: $L_{\hat{y}_{\text{spike}}}$.
    Multiply spike loss with weight λ_T.
    for task τ in (T-1) **do**
      Calculate loss for each τ: $L_{\hat{y}_{\text{voltage}}}. \in \mathbb{R}$
      Multiply loss with weights λ_τ.
      if batch = 0 **then**
        Set initial losses: $L_{(0,i)} \in \mathbb{R}$.
      end if
      Updated task weights: $\lambda_\tau = \left(\frac{L_{\hat{y},\lambda}}{L_{0,i,\lambda}} \right)^{\alpha_b}$.
    end for
    Calculate total loss: $L_{\hat{y}} = L_{\hat{y}_{\text{voltages}}} \cdot \lambda_\tau + L_{\hat{y}_{\text{spike}}} \cdot \lambda_T$, where $\tau \in (T-1)$
  end for
end for

tiply it with the weights. The tasks that perform poorly in our network have ratios $\frac{L_{\hat{y},\lambda}}{L_{0,i,\lambda}}$ close to 1 and contribute more to the gradient and total loss of our network. On the contrary, we have that tasks with ratios close to 0 contribute less to the overall loss and gradient. The balancing parameter α_b decides the influence of the task-specific weights, and if we were to adjust α_b towards 0; the LBTW would essentially approach the standard MTL [43]. We will be using $\alpha_b = 0.5$ in our neural network.

Task-balancing methods are ideal when certain tasks are sufficiently related. In our case, the compartmental potentials in proximity should be highly correlated, and the far-apart compartments may have larger diversity. We also have a binary classification task, the spiking prediction. Theoretically, the spiking task could be subjected to negative transfer since it is not a regression task like the compartmental potential values. It is interesting to examine if implementing the LBTW method improves the spike prediction in our MTL models.

3.9 Diversity

It is interesting to study the diversity of the experts in our soft-parameter MTL models. We will therefore be implementing a diversity measure so that we can compare the diversity score of both the MMoE and the MMoEEX. The diversity can be represented as a distance d^e between the experts output $f_o \forall o \in [0,1,2,.....,N]$, where N is the number of samples. If we consider the experts i and j we get:

$$d^e_{i,j} = \sqrt{\sum_{o=0}^{N} \left(f_i(x_o) - f_j(x_o)\right)} \tag{3.22}$$

Where f_i and f_j are the learned representations of the output from expert i and j. We keep all our distances $d^e_{i,j}$ in a matrix $D \in \mathbb{R}^{N \times N}$ We now want to scale the distances $d^e_{i,j}$ so they are in the range [0,1], which is done by dividing the distances in D by the maximum distance in D. We use the mean entry in D as our diversity score $\bar{d}^e$. We have that experts with $d_{i.j}$ values close to 0 are almost identical and experts with $d_{i.j}$ values close to 1 are very diverse.

3.10 Implementation

3.10.1 Implementation and Dependencies

In this master thesis, we built upon existing code created by Jonas Verhellen and Kosio Beshkov. The initial code included the implementation of the multi-task soft-parameter models (MMoE and MMoEEx). In this thesis, we implemented the hard-parameter method (MH) and the LBTW task-balancing algorithm. Our MTL models are implemented in Python with the PyTorch and numpy extensions. In addition, we use the Pytorch Lightning wrapper and Hydra configuration files to streamline the deep learning training process. The result figures in this thesis are made with Matplotlib and Seaborn. We wrote code that can run on both CPU and GPUs. The code will be available on GitHub: https://github.com/Jonas-Verhellen/LFDeep.

3.10.2 Training and Tuning

The deep learning neural networks were trained on GPUs from the ML Nodes cluster, provided by the University of Oslo [44]. We trained our models using four RTX2080Ti cores on a machine equipped with 32 cores (AMD) and 128 RAM (GiB). Our computational resources were somewhat limited during this project due to a concurrent course at UiO having the priority usage of the machine we were using. This constraint limited us to only running the models for 300 epochs; however, the loss values did seem to stabilise before this number. In an upcoming publication by Jonas Verhellen, Kosio Beshkov, Torbjørn V. Ness, Sebastian Amundsen and Gaute T. Einevoll, the models will be trained for more than 300 epochs. The MH used ca. 3.15 seconds per iteration (s/it), MMoE used ca. 4.03 s/it, and MMoEEx used ca. 4.04 s/it when running on GPUs. For the MMoEEx, we utilized a probability exclusivity of 0.1, we also tried to use 0.5, but it turned out to be too high, given that the loss flatlined after a couple of epochs.

Chapter 4

Results and Discussion

Our MTL methods undergo training for approximately 300 epochs, achieving stable loss values quite a bit before this point. Considering this stabilisation of losses and the constraints of our computational resources (see Section 3.10.2), we focus our analysis on models trained for 300 epochs. We have produced results for the models with and without the LBTW task-balancing mechanism introduced in Section 3.8. The models without task balancing are referred to as standard models, while the models with task balancing are called balanced models. We intend to find the MTL model that performs best in terms of task prediction. It is possible to employ a plethora of different metrics to evaluate the accuracy of a model. However, we will mainly focus on loss scores for the models and diversity metrics for the different experts.

We will compare our MTL models by analysing the progression of losses for the balanced and standard model archetypes. We will create a plot displaying both the raw data of loss values and a smoothed version for easier interpretation. The smoothed function we are using is called ”exponential weighted moving” (emw) [45]. It is also interesting to figure out which models perform best for specific task predictions, which can be accomplished by studying the losses for individual compartments. In addition, we will study the diversity of the experts in MMoE and MMoEEx for both the standard and balanced models; this will give us insight into how well our models capture diverse data characteristics. We will also combine these diversity metrics with

weight plots, which visualise which experts contribute most to specific tasks.

4.1 Loss Metric Evaluation

4.1.1 Standard Models

For the standard models, both MH and MMoEEx fluctuate around the same training loss values around 0.50, while the MMoE model has a slightly lower training loss, fluctuating around 0.40; see Figure 16; this indicates that the MMoE has learned the patterns in the training data more effectively than the MH and MMoEEx model. However, it is important to note that this does not necessarily mean that the soft-parameter models perform better than the hard-parameter model on unseen test data; therefore, we must evaluate the models on a separate validation data set.

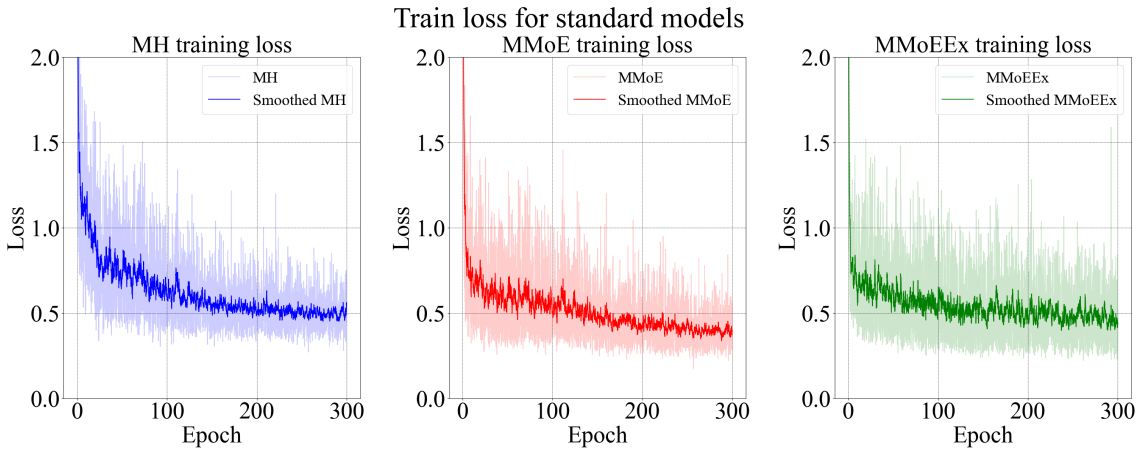

Figure 16: The training loss for the standard MH, MMoE and MMoEEx models.

Sometimes there is a discrepancy between training loss and validation loss due to overfitting, which is when the model starts to fit the noise of the data. We want a model that is neither too simplistic, failing to capture the underlying patterns in the data, and not too complex, leading to the fitting of noise in the data set [46]. For the validation data set, MMoE and MMoEEx stabilise at losses around 0.60 and 0.57, respectively, and the MH model seems to stabilise

at around 0.72; see Figure 16. Of the standard models, MMoEEx seems to perform the best, given that this method has the lowest validation loss. However, there are indications that the MMoE outperforms MMoEEx for longer runs, supported by unpublished work by Jonas Verhellen, Kosio Beshkov, Torbjørn V. Ness, Sebastian Amundsen and Gaute T. Einevoll.

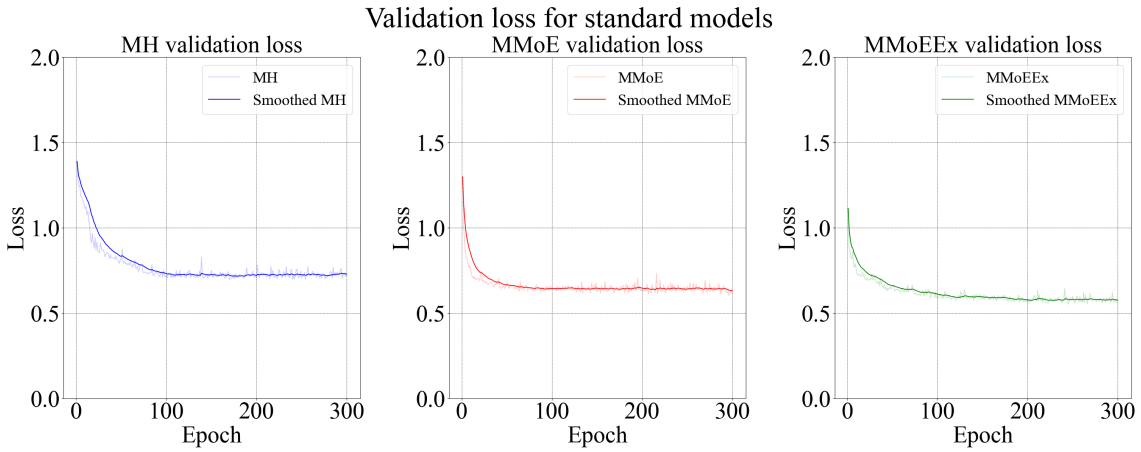

Figure 17: The validation loss for the standard MH, MMoE and MMoEEx models.

Generally, the validation loss values are larger than the training losses, which could indicate that we are slightly overfitting our models. It could be that our models fit noise and outliers, as well as the underlying patterns of the data; this could negatively impact how well our models perform on unseen test data. However, it should be mentioned that since our models are trained on the training set, they are expected to show somewhat superior performance on this data compared to the validation set.

4.1.2 Balanced Models

We also trained networks using the LBTW task-balancing algorithm (see Section 3.8). For the balanced models, MH and MMoEEx fluctuate around 0.50, and MMoE has a training loss just under 0.50; see Figure 19. Here the training loss for the balanced MH and MMoEEx models seems to be similar to the standard MH and MMoEEx models. In addition, the training loss for MMoE seems to be slightly higher for the balanced archetype than for the standard

archetype. In Figure 19, we have the validation loss for the balanced models. The validation loss stabilises at around 0.75 for the balanced MH, which is slightly larger than the loss from the standard MH. The validation loss for the balanced MMoE and MMoEEx stabilised around 0.65, which is higher than the validation loss obtained for the standard models.

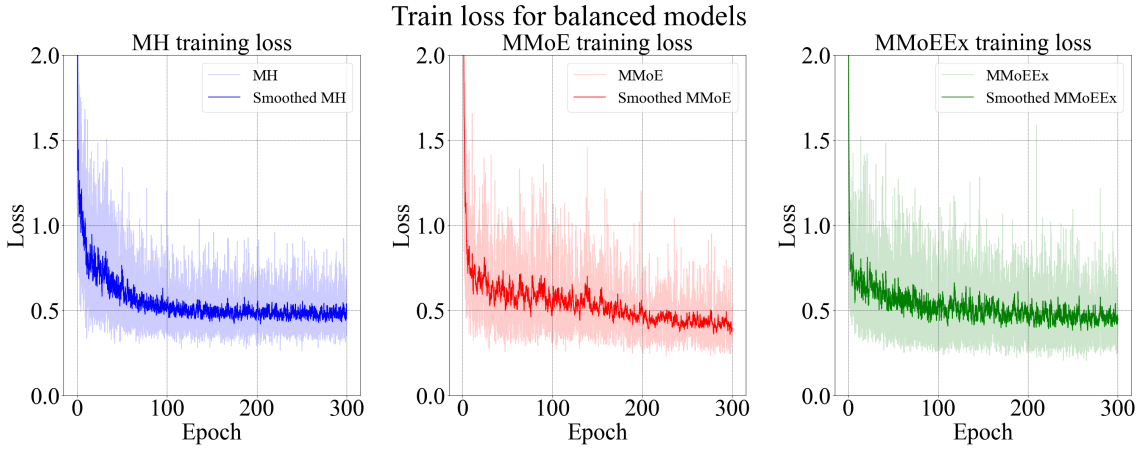

Figure 18: The training loss for the balanced MH, MMoE and MMoEEx models.

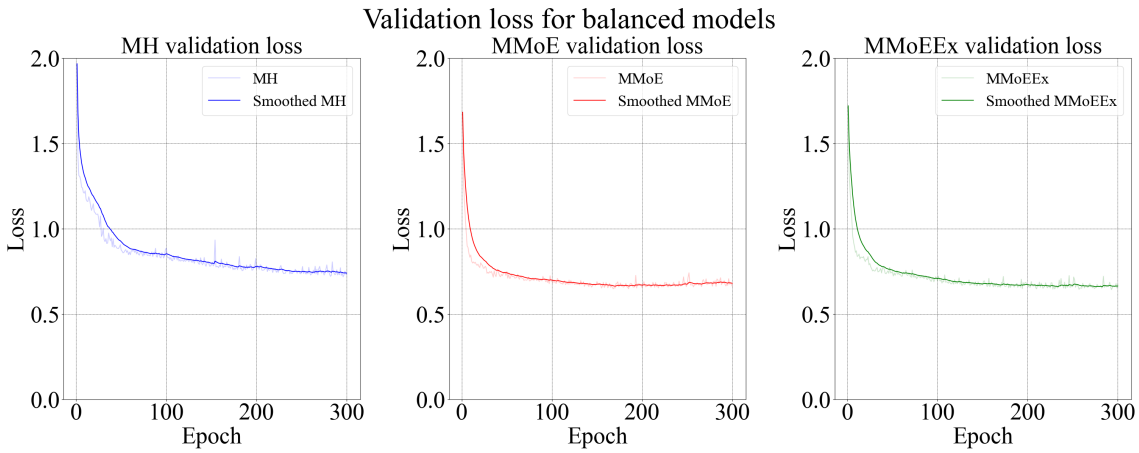

Figure 19: The validation loss for the balanced MH, MMoE and MMoEEx models.

The validation losses for our MTL methods suggest that MMoE and MMoEEx exhibit the strongest performance in predicting tasks. Whereas the MH model exhibits somewhat subpar performance, as evidenced by its higher loss values than the soft-parameter models. There was only a minimal difference in

performance between the standard and balanced models, where the balanced models had slightly larger loss values compared to the standard models for all MTL methods. It is important to note that evaluating the loss is only one measure of assessing how well models perform. An additional measure is the diversity metric (see Section 4.3), which can be used to assess the feature learning capabilities of our experts in MMoE and MMoEEx.

4.2 Compartmental Losses

Our MTL methods will exhibit varying performance levels in predicting the voltages of the different compartments in our neuron model. We will study the compartmental validation losses to understand how accurately our MTL methods predict the neurons' different sections. We can divide the neuron model into three main sections: the basal, oblique and apical parts of the neuron [8]. By tracking the compartmental losses, we can determine if certain MTL methods are more suited for simulating specific sections of the neuron. This classification of methods can be helpful if the goal is to prioritize the prediction of a specific neuron section, as one can select the model that performs best for that part of the neuron. However, in this thesis, we are primarily interested in finding the most accurate MTL method across all three sections of the neuron.

4.2.1 Loss Distribution for MTL Models

The sections of the neuron model are depicted with the colours blue, orange and green corresponding to the basal, oblique and apical area, respectively; see Figure 20 for the standard models. Generally, the apical and oblique compartments have the lowest loss values, and the basal compartments have the largest loss values. Most compartments have losses lower than 2, but a few outliers have higher loss values than this. The MH seems to have most of these outliers, while the MMoEEx has the least.

In Figure 21, we have the compartmental losses for the balanced models.

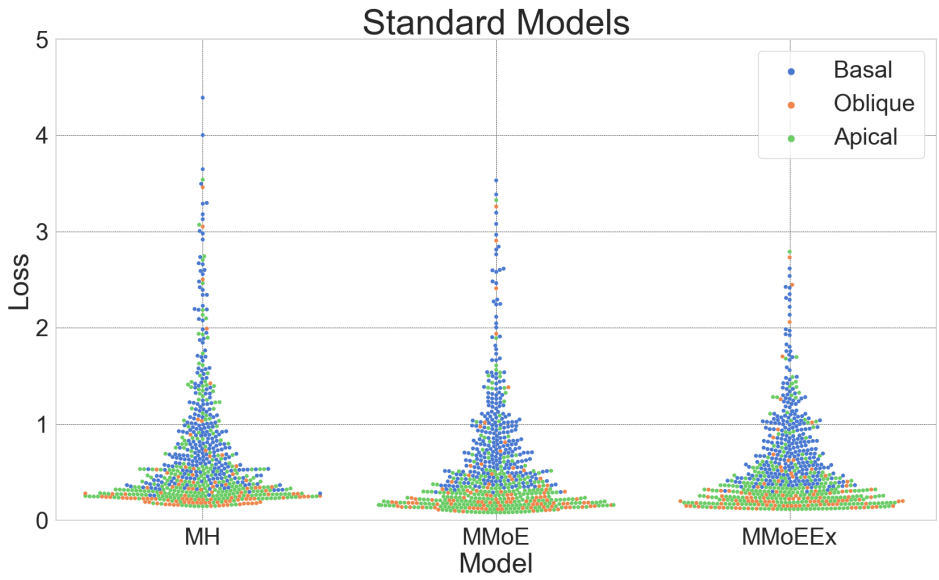

Figure 20: Depicted are the compartmental voltage losses for the standard MH, MMoE and MMoEEx. Generally, the apical and oblique compartments have the lowest loss values, while the basal compartments have the highest losses. By looking at the figure, it seems that MMoEEx has the lowest loss values across compartments.

The apical and oblique compartments are generally the most accurate here, while the basal compartments typically have the highest loss values. For the balanced models, the MTL methods perform more similarly to each other than the standard models, and it is hard to differentiate which of the MTL methods performs the best. Generally, the compartmental losses are higher for the balanced models than the standard ones, especially for the MMoEEx.

Looking at Figure 20 and 21, it seems that the standard MMoEEx outperform the other models concerning compartmental losses. However, it can be challenging to determine the performance of the models regarding the specific sections of the neuron by only looking at the figures above; we will therefore look at plots with the area of the neuron on the x-axis and the loss values on the y-axis in the upcoming section. These plots will hopefully give us deeper insight into how our methods perform for the different parts of the neuron.

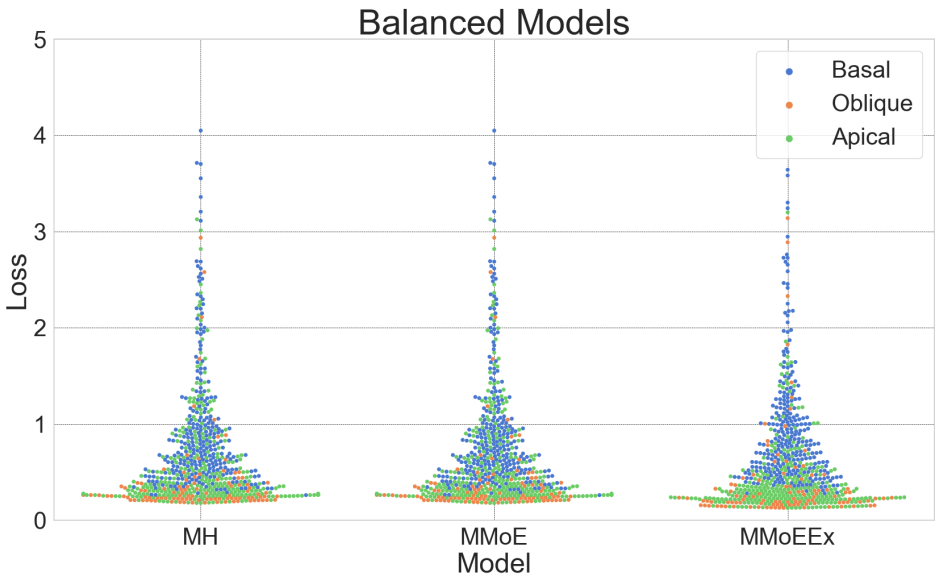

Figure 21: Depicted are the compartmental voltage losses for the balanced MH, MMoE and MMoEEx. Generally, the apical and oblique compartments have the lowest loss values, while the basal compartments have the highest losses. The MTL methods seem to perform pretty similarly across compartments, and it is hard to differentiate the best-performing model by visually analysing the figure.

4.2.2 Comparison of Standard and Balanced Models

In Figure 22, 23 and 24, we have plotted the compartmental losses for each section of neuron given by the standard and balanced MTL methods. The figures are not all that different when considering the shape of the swarm plots. Generally, the basal section has the highest loss values, while the apical and oblique sections have the lowest, in agreement with our observations from Section 4.2.1. In addition, we can see fewer compartments in the oblique area than in the basal and apical sections of the neuron. The standard model compartments are depicted in blue, red and green for the MH, MMoE and MMoEEx, respectively, and the balanced model compartments are displayed as yellow for the MH, light blue for the MMoE and pink for the MMoEEx.

We can see that the loss values for the standard MH (Figure 22) are generally higher for the balanced MH than for the standard MH when considering the apical and oblique sections of the neuron. However, the compartments in

the basal section of the neuron perform pretty similarly for both the balanced and standard MH. We introduce the term ”outlier compartmental loss values” to describe the compartments with elevated loss values compared to the typical compartmental losses. These outlier compartments have larger loss values for the standard MH than the balanced MH. These larger loss values could come from the fact that the balancing mechanism constrains the task predictions so that no single tasks dominate the learning process, which may lead to less variation in loss values across compartments.

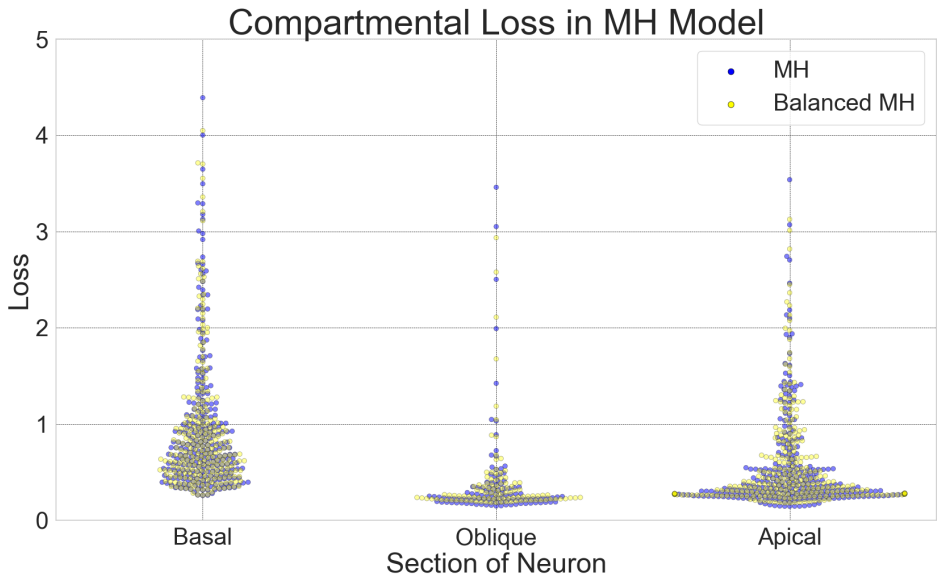

Figure 22: Depicted are the compartmental losses given by the different sections of the neuron for the MH method. The compartments for the standard MH are represented as blue dots, while the compartments for the balanced MH are depicted as yellow dots. We can see that the compartmental loss values for the basal section of the neuron perform similarly for the balanced and standard MH. For the oblique and apical sections of the neuron, it seems that the standard MH outperforms the balanced MH.

The compartmental loss values for the MMoE method (Figure 23) are similar to those of the MH. Again, the standard models perform better than the balanced models for the oblique and apical sections of the neuron, while the standard and balanced MMoE perform similarly for the basal neuron section. Interestingly, the outlier compartmental loss values are generally higher for the balanced MMoE than for the standard MMoE, and there seem to be more

outlier compartments for the balanced MMoE. This observation goes against our previous proposal, namely that the balancing mechanism forces less variation across compartmental losses.

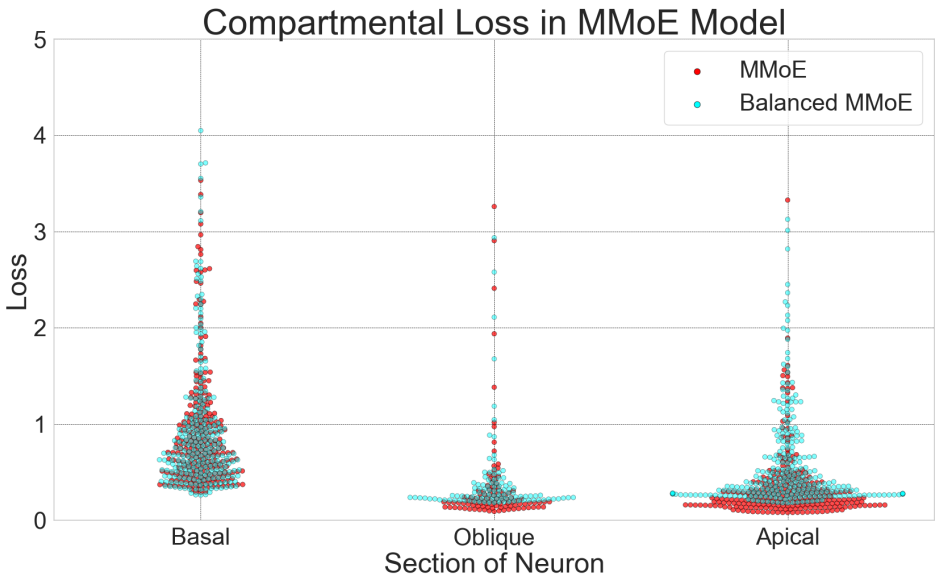

Figure 23: Depicted are the compartmental losses given by the different sections of the neuron for the MMoE method. The compartments for the standard MMoE are represented as red dots, while the compartments for the balanced MH are represented as light blue dots. The compartmental loss values are generally higher for the balanced MMoE than for the standard MMoE for the oblique and apical sections of the neuron; the compartmental losses are more similar in the basal section for both MMoE archetypes.

The compartmental losses are more similar for the standard and balanced MMoEEx (Figure 24) compared to the other MTL methods. Here the balanced MMoEEx seems to outperform the standard MMoEEx for both the basal and oblique sections of the neuron. However, the balanced MMoEEx seem to have slightly more outlier compartments than the standard MMoEEx and the compartmental losses for the balanced MMoEEx still appear larger than the standard MMoEEx for the apical section.

The results from Section 4.2.1 and 4.2.2 indicate that the balanced MTL methods have larger compartmental losses than the standard MTL methods, specifically in the oblique and apical sections of the neuron. The balanced models only seemed to reduce the number of outlier compartments in the MH

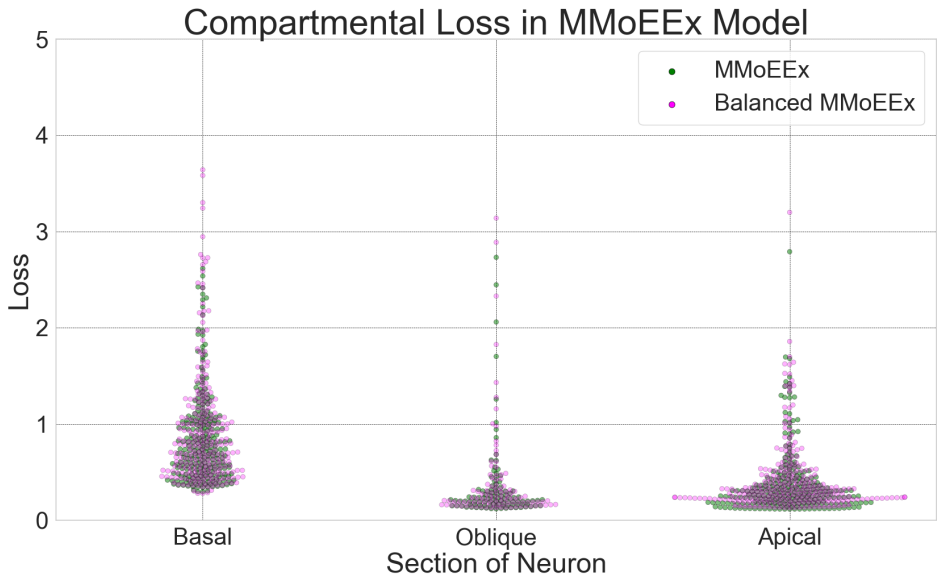

Figure 24: Depicted are the compartmental losses given by the different sections of the neuron for the MMoEEx method. The compartments for the standard MMoEEx are represented as green dots, while the compartments for the balanced MH are depicted as pink dots. Here the balanced MMoEEx seems to outperform the standard MMoEEx for the basal and oblique sections of the neuron. In contrast, the standard MMoEEx seems more accurate than the balanced MMoEEx for the compartments in the apical section of the neuron.

model, but we found no such correlation for MMoE or MMoEEx. Our findings indicate no concrete improvement in compartmental losses by implementing the LBTW task-balancing mechanism (see Section 3.8). In this specific run of simulations, the model with the lowest compartmental loss values was the standard MMoEEx model. It is important to note that all models were trained on 300 epochs, and the results could be different if we let our simulations go on for longer.

4.3 Diversity

In Section 3.9, we introduced a method to calculate the diversity of the experts in our soft-parameter models. The diversity metric can help us understand how well our models capture diverse data characteristics. We calculated the diversity score given by equation 3.22, as well as the determinant and per-

manent of the distance matrix D between experts (see Section 3.9). The definition of the permanent operation is almost the same as the determinant, the only difference being that for the permanent operation, the entries in the matrix are added when summing the different elements in the matrix, while the determinant operation alternates between addition and subtraction when totalling the elements in the matrix.

4.3.1 Standard Models

The diversity score for the standard MMoE and MMoEEx is given in Figure 25. We can see that the diversity score for the MMoE is initially at its maximum and begins to decrease until approximately 70 epochs. Afterwards, it increases again, reaching a smaller peak at around 150 epochs. Finally, the diversity decreases again and eventually stabilises after around 225 epochs. The diversity score for the MMoEEx demonstrates fluctuations near its maximum value during the initial phase and after 100 epochs, with a noticeable dip transpiring between these two periods. The final stabilised diversity fluctuates around 0.59 for the MMoE and 0.64 for the MMoEEx.

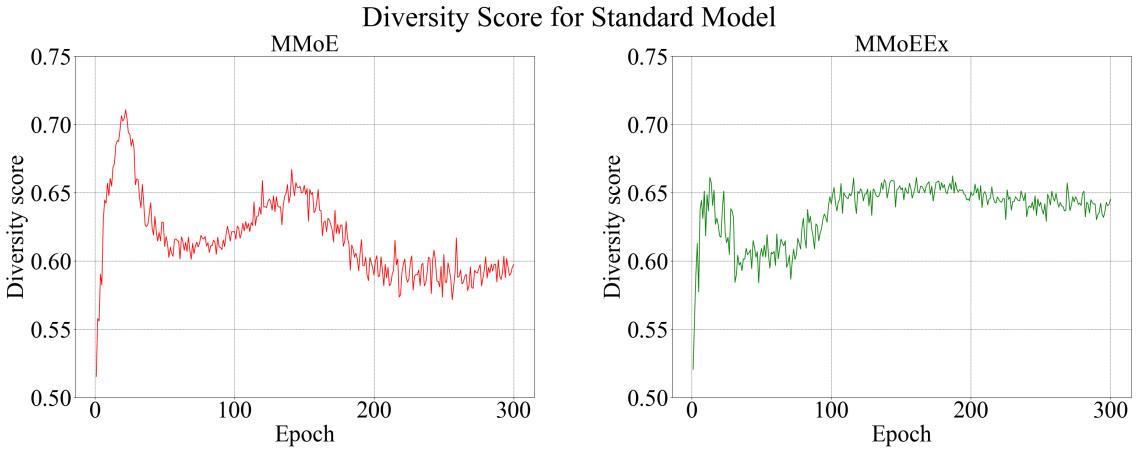

Figure 25: The diversity score for the standard soft-parameter models.

We see a similar pattern for the diversity determinant as we did for the diversity score; the only difference is that the shape is more pronounced for the diversity determinant than for the diversity score; see Figure 26.

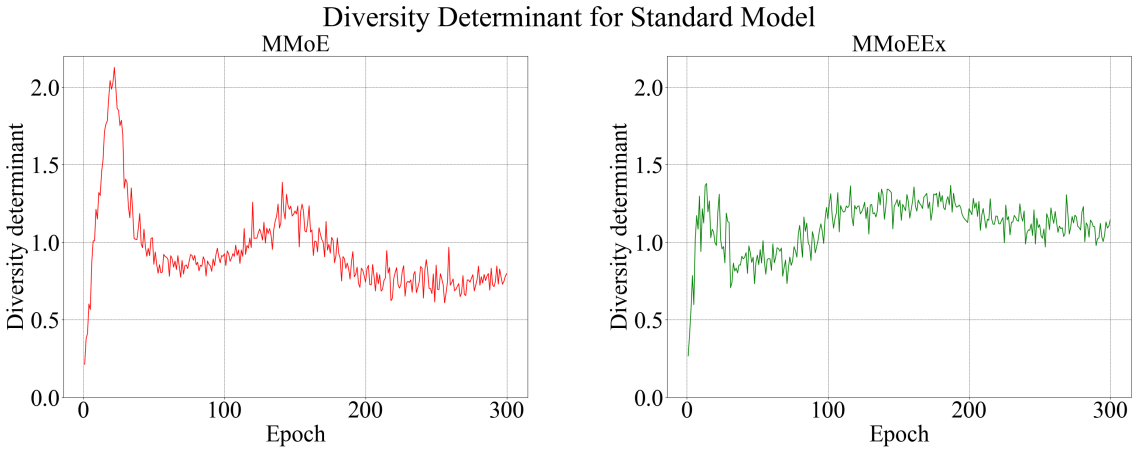

Figure 26: The diversity determinant for the standard soft-parameter models.

The shape of the diversity permanent is as good as identical to the diversity determinant; see Figure 27. However, the scale between the determinant and permanent is evidently different, where the diversity permanent values are significantly larger than the diversity determinant values; this is most likely due to the fact that the permanent diversity metric does not include negative values in its calculations while the diversity determinant does.

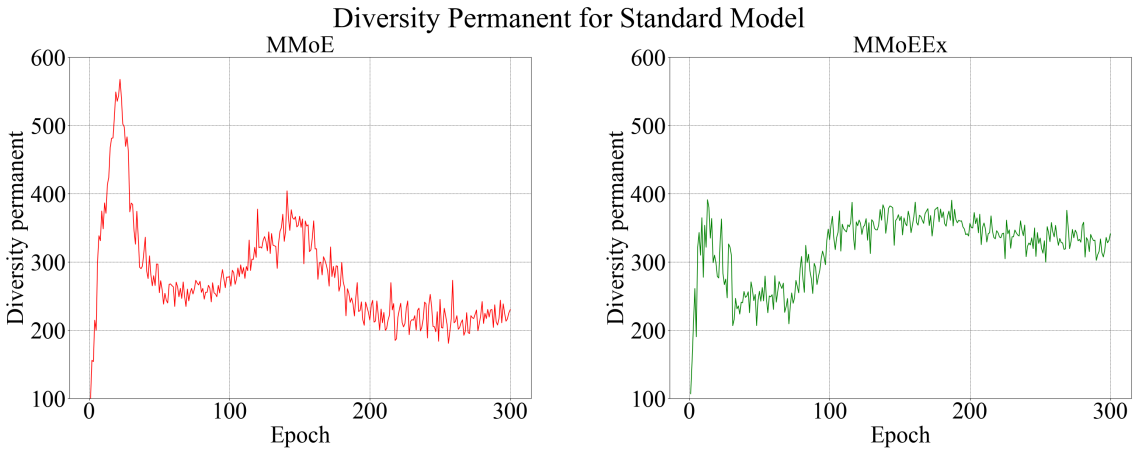

Figure 27: The diversity permanent for the standard soft-parameter models.

4.3.2 Balanced Models

We also calculated the diversity for the balanced models. Figure 28 gives the diversity score for the balanced models. The diversity score for the MMoE increases until 50 epochs, where the validation score is at its peak. Following

this, the validation score decreases and reaches a stabilisation point, fluctuating just under 0.60. The MMoEEx has a stable increase in diversity score until around 60 epochs. Afterwards, it stabilises at just below 0.50. The diversity score is generally lower for the balanced models than for the unbalanced models.

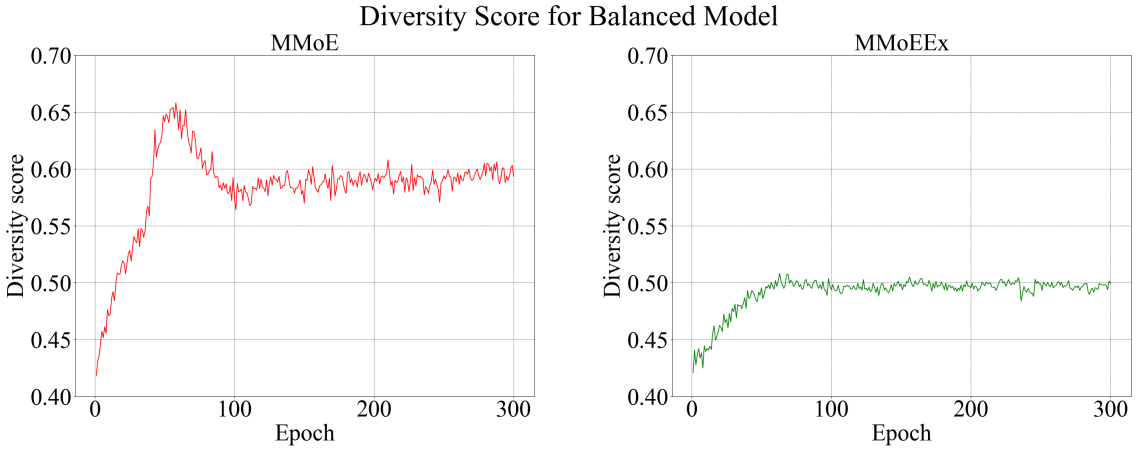

Figure 28: The diversity score for the balanced soft-parameter models.

The shape of the ”diversity determinant curves” are similar to that of the diversity score; see Figure 29. However, the diversity determinant for the balanced MMoE and MMoEEx are surprisingly different in relation to scale; the MMoEEx stabilises at a much lower value than the MMoE; this is not consistent with our observations from the standard models, where the diversity determinant is similar in terms of stabilised magnitude for both MMoE and MMoEEx.

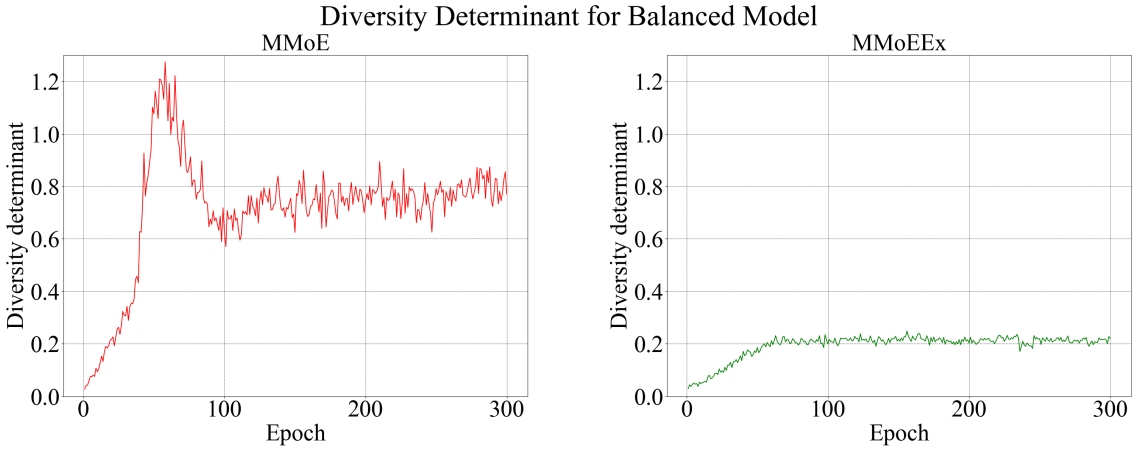

Figure 29: The diversity determinant for the balanced soft-parameter models.

In Figure 30, we can see the diversity permanent for the balanced MMoE and MMoEEx. The shape of the permanent metric curve is basically identical to the shape of the determinant metric curve. Again, the magnitude of the MMoE diversity is considerably larger than the MMoEEx.

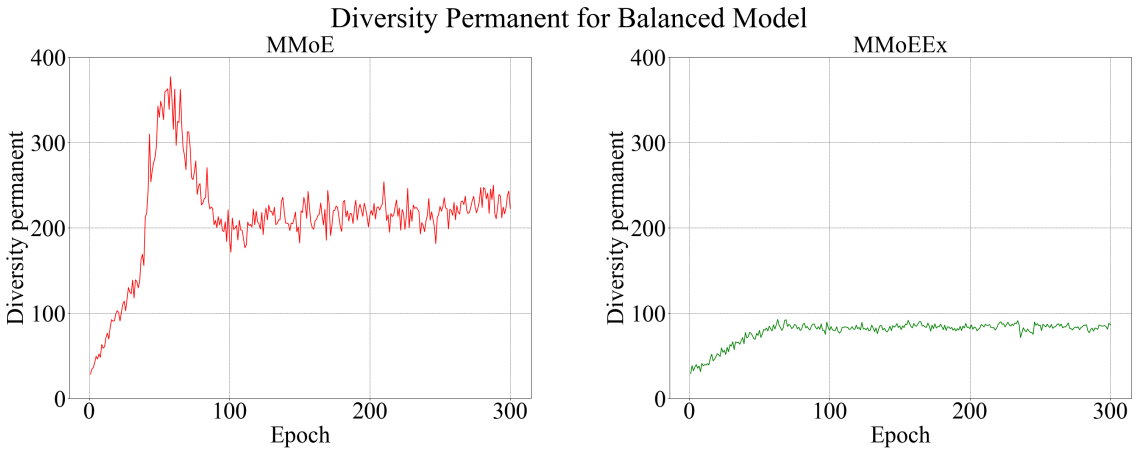

Figure 30: The diversity permanent for the balanced soft-parameter models.

Generally, the diversity is higher for the standard models than the balanced models, especially for the MMoEEx method; this can be explained by the fact that in a balanced model, the tasks are all ”treated fairly”, meaning that no tasks are supposed to dominate the learning process, which could perhaps introduce a more similar prediction behaviour between experts, consequently making them less diverse. In contrast, the unbalanced models may overfit some tasks and underfit others, which could lead to certain experts specialising in specific overfit and underfit tasks, which could contribute to greater diversity among experts.

Our observations seem to indicate that the standard models are more adept at creating diverse representations of the data within their experts. Despite this, it appears that we do not require that much diversity to accurately predict our tasks, supported by the fact that the balanced models with lower diversity performed almost up to par with the standard models with considerably higher diversity when we calculate the losses (see Section 4.1).

4.4 Task Contribution from Experts

It is interesting to study which experts contribute most to various tasks. The gates in our networks are multiplied by the expert weights. They are essentially ”picking” the most relevant experts for specific tasks. We can plot the weight distribution of the gates against the tasks to gain insight into which experts contribute most to the different tasks. Figure 31 shows the gate weight distribution for the standard models. Here we use E1, E2, E3, E4 and E5 as abbreviations for expert 1, expert 2, expert 3, expert 4 and expert 5, respectively. It is apparent that the experts specialise in different tasks; for the MMoE model, E1 and E5 contribute most to the later tasks, while E2 specialise more in the earlier tasks. E3 seems to specialise more for tasks in the middle. Interestingly, E1, E2, E3 and E4 seem to target the spiking task, as evidenced by the sharp weight increase around task 641, which is the spiking task. MMoEEx has similar weight distributions to MMoE for E1, E3 and E4, while E2 focuses more on later tasks and E5 targets the earlier tasks and the spiking task.

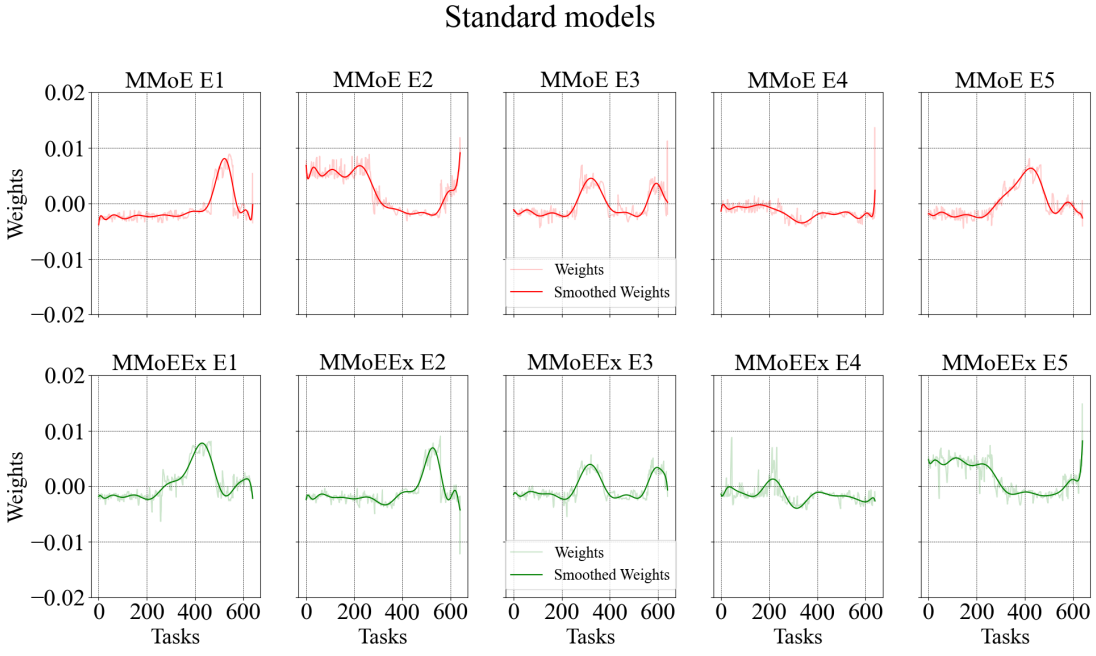

Figure 31: Weight distribution for standard models.

The weight distributions for the balanced models are given in Figure 32. The weight distributions for the balanced MMoE are really similar to that of the standard MMoE. However, the balanced MMoEEx have a significantly different weight distribution than the standard MMoEEx for E1 and E5; it almost looks like these two experts have interchanged weight distributions compared to the standard MMoEEx. There also seems to be more variability in weight values for the balanced MMoEEx compared to the standard MMoEEx in all experts, as evidenced by the distance between the weights and smoothed functions.

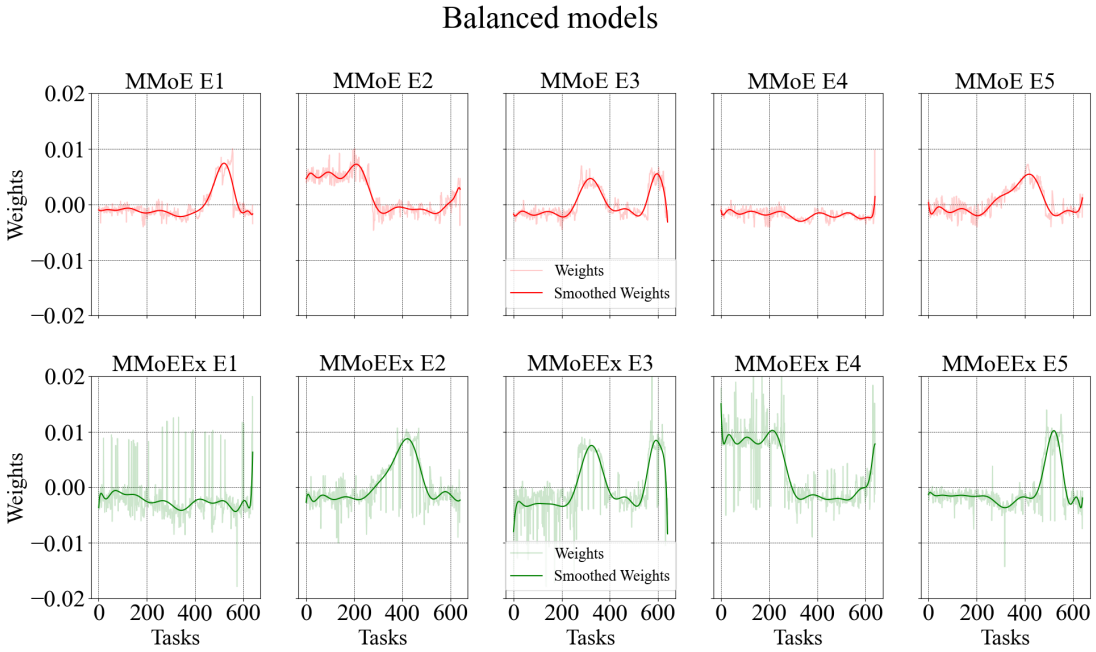

Figure 32: Weight distribution for balanced models.

The weight distribution plots could maybe indicate that our tentative theory in 4.3 could hold some merit, the hypothesis was that the balancing mechanism would facilitate more similar prediction behaviour among the experts. We can see that the weight data is more ”scattered” in the balanced models than the standard models, primarily for the MMoEEx, which could mean that the experts are more generalized to a greater number of tasks, essentially making our experts less diverse. The results from Section 4.3 and 4.4 seem to indicate that the standard models are generally more diverse than the bal-

anced models, especially for the MMoEEx model.

4.5 Spike Prediction

The main reason for implementing the balancing mechanism in our models was to attempt to predict the spike initiation task. However, we did not manage to predict spike initiation for any of our models. The spiking task is likely too different from the other compartmental voltage tasks for our MTL methods to predict it accurately. Interestingly, some of the experts in Figure 31 and 32 do have significant weight values for the spike prediction task (task 641), but it was not enough to generate accurate spike predictions. It could very well be that the spike prediction task is too dissimilar compared to the compartmental voltage predictions for our MTL method to predict it accurately; however, there may be some structural changes we could make to our networks in order to improve the spiking task's predictability.

Implementing a task-balancing mechanism other than the LBTW may improve the ability of our networks to predict the spiking task. For example, in [11], they implement a task-balancing approach called MAML [47], which perhaps could improve the spike prediction capabilities of our networks. It could also be beneficial to increase the depth of our experts in order to improve task-prediction accuracy. In Section 3.3, we introduced a residual learning framework [37], which may be appropriate to incorporate if we add more depth to our experts.

4.6 Model Selection

We have tested our MTL methods by looking at loss and diversity metrics, and the methods were tested with and without the LBTW task-balancing mechanism. Overall, we found that the soft-parameter models without task balancing had the lowest validation losses for the whole neuron and the compartmental loss values corresponding to the different sections of the neuron. We

found that the balancing mechanism did not improve the compartmental voltage or spike predictions; this means that the task-balancing mechanism can effectively be disregarded for our specific research question. The standard soft-parameter models' diversity was similar, and the balanced archetypes had significantly lower diversity than the standard models. Even though the standard MMoEEx outperformed the standard MMoEE in terms of accuracy (lower loss values) for 300 epochs, unpublished work by Jonas Verhellen, Kosio Beshkov, Torbjørn V. Ness, Sebastian Amundsen, and Gaute T. Einevoll indicates that the MMoE model outperforms MMoEEx for longer runs. The optimal MTL method for our research question would be the standard soft-parameter sharing models, either the MMoE or the MMoEEx.

Chapter 5

Conclusion and Future Work

5.1 Conclusion

In this thesis, we implemented three MTL methods of different complexity to predict the electrical activity of a compartmental L5b PC model [6]. We employed a multi-task hard-parameter sharing model, MH, and two multi-task soft-parameter sharing models, MMoE and MMoEEx. This study aimed to map the input-output relationship from the L5b compartmental neuron model using our proposed MTL methods in a computationally efficient way; we wanted to predict both the spiking behaviour and the compartmental voltages of the neuron model using DNNs instead of having to calculate the series of cable equations and model dynamics found in biophysical neuron models.

Using our MTL methods, we successfully recreated the input-output relationship from the compartmental L5b PC model [6]. However, the deep learning models struggled to predict the neuron model's spiking behaviour; therefore, we decided to implement the LBTW task-balancing mechanism to improve our networks' prediction capabilities. Unfortunately, the task-balancing mechanism did not ameliorate the spiking prediction capabilities of the networks. Consequently, we primarily concentrated on assessing the performance of the MTL approaches concerning predicting the compartmental voltages within the neuron model.

We compared the standard hard-parameter approach (MH) with the two

standard soft-parameter methods (MMoE and MMoEEx) by employing loss and diversity metrics. Additionally, we assessed the performance of the balanced MTL archetypes by the same metrics. The standard MMoEEx method exhibited the lowest total loss out of our MTL methods while having the fewest outlier compartments concerning loss. The MH model exhibited the highest loss out of our MTL methods; however, it is worth mentioning that this model was the fastest to simulate and, consequently, the most computationally efficient model. Overall, the models with the task-balancing mechanism performed worse than the standard approaches concerning loss values. The diversity metric was similar for the standard soft-parameter models, while the balanced models had significantly lower diversity, especially the balanced MMoEEx. Interestingly, all the soft-parameter models designated some of their experts to fit the spiking task, but it was not enough to predict the spiking behaviour of the neuron.

Generally, the balanced MTL methods performed worse than the standard counterparts for both loss and diversity. Therefore, it is safe to assume that the LBTW task-balancing algorithm is not directly applicable to our research question. Our results indicate that the optimal MTL method for our project would be the standard MMoEEX model. However, it is worth mentioning that we only trained our networks for 300 epochs, and we may very well obtain different results if we let our simulations go on for longer; unpublished work by Jonas Verhellen, Kosio Beshkov, Torbjørn V. Ness, Sebastian Amundsen and Gaute T. Einevoll indicate that the standard MMoE outperforms MMoEEx for longer runs. Based on this, it is safe to assume that either standard MTL soft-parameter model is best suited for predicting compartmental voltages from the L5b PC model.

5.2 Further Research

The MTL framework from this project can be used to effectively model compartmental voltage progression of biophysical advanced neuron models, not

limited to the compartmental L5b PC model; our MTL framework can be extended to model other advanced neuron models, simultaneously reducing the computational resources required for their simulation. In future research, it would be intriguing to explore the possibility of implementing several distilled neuron models into networks in order to simulate sections of the brain. We could, for example, implement our MTL framework to something like the Allen V1 brain models for the primary visual cortex [48] and consequently reduce the time it takes to run these models. The work done in this thesis can significantly improve the computational efficiency in estimating synaptic currents and local field potentials for neuron models when contrasted with conventional modelling techniques.

Bibliography

[1] James J. Jun et al. “Fully integrated silicon probes for high-density recording of neural activity”. In: *Nature* 551.7679 (2017), pp. 232–236. DOI: 10.1038/nature24636.

[2] Dorian Krause. “JUWELS: Modular Tier-0/1 Supercomputer at the Jülich Supercomputing Centre”. In: *Journal of large-scale research facilities JLSRF* 5 (2019), A135–A135. DOI: 10.17815/jlsrf-5-171.

[3] Shanyu Chen et al. “How Big Data and High-performance Computing Drive Brain Science”. In: *Genomics, Proteomics & Bioinformatics* 17.4 (2019), pp. 381–392. DOI: 10.1016/j.gpb.2019.09.003.

[4] A. L. Hodgkin and A. F. Huxley. “A quantitative description of membrane current and its application to conduction and excitation in nerve”. In: *The Journal of Physiology* 117.4 (1952), pp. 500–544.

[5] Konstantinos-Evangelos Petousakis, Anthi A. Apostolopoulou, and Panayiota Poirazi. “The impact of Hodgkin–Huxley models on dendritic research”. In: *The Journal of Physiology* (2022). in press. DOI: 10.1113/JP282756.

[6] Etay Hay et al. “Models of Neocortical Layer 5b Pyramidal Cells Capturing a Wide Range of Dendritic and Perisomatic Active Properties”. In: *PLOS Computational Biology* 7.7 (2011), e1002107. DOI: 10.1371/journal.pcbi.1002107.

[7] Panayiota Poirazi, Terrence Brannon, and Bartlett W. Mel. “Pyramidal neuron as two-layer neural network”. In: *Neuron* 37.6 (2003), pp. 989–999. DOI: 10.1016/s0896-6273(03)00149-1.

[8] David Beniaguev, Idan Segev, and Michael London. “Single cortical neurons as deep artificial neural networks”. In: *Neuron* 109.17 (2021), 2727–2739.e3. DOI: 10.1016/j.neuron.2021.07.002.

[9] Richard A. Caruana. “Multitask Learning: A Knowledge-Based Source of Inductive Bias”. In: *Machine Learning Proceedings 1993*. San Francisco (CA): Morgan Kaufmann, 1993, pp. 41–48. DOI: 10.1016/B978-1-55860-307-3.50012-5.

[10] Jiaqi Ma et al. “Modeling Task Relationships in Multi-task Learning with Multi-gate Mixture-of-Experts”. In: *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. London, United Kingdom: ACM, 2018, pp. 1930–1939. DOI: 10.1145/3219819.3220007.

[11] Raquel Aoki, Frederick Tung, and Gabriel L. Oliveira. “Heterogeneous Multi-Task Learning With Expert Diversity”. In: *IEEE/ACM Transactions on Computational Biology and Bioinformatics* 19.6 (2022), pp. 3093–3102. DOI: 10.1109/TCBB.2022.3175456.

[12] David Sterratt et al. *Principles of Computational Modelling in Neuroscience*. Cambridge, MA, USA: Cambridge University Press, 2011.

[13] Wulfram Gerstner et al. *Neuronal Dynamics: From Single Neurons to Networks and Models of Cognition*. Cambridge, MA, USA: Cambridge University Press, 2014.

[14] Eric R. Kandel et al. *Principles of Neural Science, 6e*. New York, NY, USA: McGraw Hill, 2021.

[15] Dale Purves et al. *Neuroscience, 3rd ed*. Sunderland, MA, US: Sinauer Associates, 2004.

[16] BruceBlaus. *Multipolar Neuron*. Wikimedia Commons. 2013.

[17] Nelson Spruston. “Pyramidal neurons: dendritic structure and synaptic integration”. In: *Nature Reviews. Neuroscience* 9.3 (2008), pp. 206–221. DOI: 10.1038/nrn2286.

[18] Maria Toledo-Rodriguez et al. “Correlation Maps Allow Neuronal Electrical Properties to be Predicted from Single-cell Gene Expression Profiles in Rat Neocortex”. In: *Cerebral Cortex* 14.12 (2004), pp. 1310–1327. DOI: 10.1093/cercor/bhh092.

[19] Nelson Phillip, Radosavljevic Marko, and Bromberg Sarina. *Biological physics: energy, information, life*. New York, NY, USA: W. H. Freeman and Company, 2014.

[20] Julian Seifter, David Sloane, and Austin Ratner. *Concepts in Medical Physiology*. Philadelphia, PA, USA: Lippincott Williams & Wilkins, 2005.

[21] K.S Cole. “Dynamic electrical characteristics of the squid axon membrane”. In: *Annu Rev Physiol* 3 (1949), pp. 253–258.

[22] K. Deb et al. “A fast and elitist multiobjective genetic algorithm: NSGA-II”. In: *IEEE Transactions on Evolutionary Computation* 6.2 (2002), pp. 182–197. DOI: 10.1109/4235.996017.

[23] Matthew E. Larkum, J. Julius Zhu, and Bert Sakmann. “A new cellular mechanism for coupling inputs arriving at different cortical layers”. In: *Nature* 398.6725 (1999), pp. 338–341. DOI: 10.1038/18686.

[24] Wilfrid Rall. “Branching dendritic trees and motoneuron membrane resistivity”. In: *Experimental Neurology* 1.5 (1959), pp. 491–527. DOI: 10.1016/0014-4886(59)90046-9.

[25] Wilfrid Rall. “Theoretical Significance of Dendritic Trees for Neuronal Input-Output Relations”. In: *The Theoretical Foundation of Dendritic Function: The Collected Papers of Wilfrid Rall with Commentaries* (2003). DOI: 10.7551/mitpress/6743.003.0015.

[26] I. Segev and W. Rall. “Computational study of an excitable dendritic spine”. In: *Journal of Neurophysiology* 60.2 (1988), pp. 499–523. DOI: 10.1152/jn.1988.60.2.499.

[27] Oscar Herreras. “Local Field Potentials: Myths and Misunderstandings”. In: *Frontiers in Neural Circuits* 10 (2016). DOI: https://doi.org/10.3389/fncir.2016.00101.

[28] Gaute T. Einevoll et al. “Modelling and analysis of local field potentials for studying the function of cortical circuits”. In: *Nature Reviews Neuroscience* 14.11 (2013), pp. 770–785. DOI: 10.1038/nrn3599.

[29] Romain Brette and Alain Destexhe, eds. *Handbook of Neural Activity Measurement*. Cambridge, MA, USA: Cambridge University Press, 2012.

[30] György Buzsáki, Costas A. Anastassiou, and Christof Koch. “The origin of extracellular fields and currents–EEG, ECoG, LFP and spikes”. In: *Nature Reviews. Neuroscience* 13.6 (2012), pp. 407–420. DOI: 10.1038/nrn3241.

[31] Carl Gold et al. “On the origin of the extracellular action potential waveform: A modeling study”. In: *Journal of Neurophysiology* 95.5 (2006), pp. 3113–3128. DOI: 10.1152/jn.00979.2005.

[32] Matti Hämäläinen. “Magnetoencephalography—theory, instrumentation, and applications to noninvasive studies of the working human brain”. In: *Reviews of Modern Physics* 65.2 (1993), pp. 413–497. DOI: 10.1103/RevModPhys.65.413.

[33] Espen Hagen et al. “Multimodal Modeling of Neural Network Activity: Computing LFP, ECoG, EEG, and MEG Signals With LFPy 2.0”. In: *Frontiers in Neuroinformatics* 12 (2018), p. 92. DOI: 10.3389/fninf.2018.00092.

[34] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. Cambridge, MA, USA: MIT Press, 2016.

[35] Shaojie Bai, J. Zico Kolter, and Vladlen Koltun. *An Empirical Evaluation of Generic Convolutional and Recurrent Networks for Sequence Modeling*. arXiv:1803.01271. 2018. DOI: 10.48550/arXiv.1803.01271.

[36] Nitish Srivastava et al. "Dropout: A Simple Way to Prevent Neural Networks from Overfitting". In: *Journal of Machine Learning Research* 15.56 (2014), pp. 1929–1958. DOI: 10.5555/2627435.2670313.

[37] Kaiming He et al. "Deep Residual Learning for Image Recognition". English. In: ISSN: 1063-6919. IEEE Computer Society, 2016, pp. 770–778. ISBN: 978-1-4673-8851-1. DOI: 10.1109/CVPR.2016.90.

[38] Diederik P. Kingma and Jimmy Ba. *Adam: A Method for Stochastic Optimization*. arXiv:1412.6980. 2017. DOI: 10.48550/arXiv.1412.6980.

[39] John Duchi, Elad Hazan, and Yoram Singer. "Adaptive Subgradient Methods for Online Learning and Stochastic Optimization". In: *The Journal of Macine Learning Research, 12:2121–2159* 12 (2011), pp. 2121–2159. DOI: 10.5555/1953048.2021068.

[40] Tijmen Tieleman and Geoffrey Hinton. "Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude". In: *COURSERA: Neural networks for machine learning* 4.2 (2012), pp. 26–31.

[41] Gido M. van de Ven, Tinne Tuytelaars, and Andreas S. Tolias. "Three types of incremental learning". In: *Nature Machine Intelligence* 4.12 (2022), pp. 1185–1197. DOI: 10.1038/s42256-022-00568-3.

[42] Michael A. Nielsen. *Neural Networks and Deep Learning*. San Francisco, CA, USA: Determination Press, 2015.

[43] Shengchao Liu, Yingyu Liang, and Anthony Gitter. "Loss-Balanced Task Weighting to Reduce Negative Transfer in Multi-Task Learning". In: *Proceedings of the AAAI Conference on Artificial Intelligence* 33 (2019), pp. 9977–9978. DOI: 10.3389/fncir.2016.00101.

[44] Machine learning infrastructure (ML Nodes). *University Centre for Information Technology, University Of Oslo, Norway*. 2023.

[45] The pandas development team. *pandas.DataFrame.ewm — pandas 2.0.1 documentation*. 2023. DOI: 10.5281/zenodo.3509134.

[46] Jake Lever, Martin Krzywinski, and Naomi Altman. "Model selection and overfitting". In: *Nature Methods* 13.9 (2016), pp. 703–704. DOI: 10.1038/nmeth.3968.

[47] Sungjae Lee and Youngdoo Son. "Multitask learning with single gradient step update for task balancing". In: *Neurocomputing* 467 (2022), pp. 442–453. DOI: 10.1016/j.neucom.2021.10.025.

[48] Yazan N. Billeh et al. "Systematic Integration of Structural and Functional Data into Multi-scale Models of Mouse Primary Visual Cortex". In: *Neuron* 106.3 (2020), 388–403.e18. DOI: 10.1016/j.neuron.2020.01.040.

Appendix

A Hodgkin and Huxley Model Dynamics

Here we present the complete set of equations for the Hodgkin and Huxley model. We introduced equation 2.6 in Section 2.3, which describes the membrane potential over a small section of the giant squid axon. We have three coupled differential equations that are used to adjust the gating variables p_m, p_h and p_n in equation 2.6. The potassium activation p_n is given by:

$$\begin{aligned} \frac{\partial p_n}{\partial t} &= \alpha_n(1-p_n) - \beta_n p_n \\ \text{where} & \\ \alpha_n &= 0.1\frac{V_s+55}{1-e^{-(V_s+55)/10}} \\ \beta_n &= 0.125e^{-(V_s+65)/80} \end{aligned} \tag{A.1}$$

The sodium activation p_m is given by:

$$\begin{aligned} \frac{\partial p_m}{\partial t} &= \alpha_m(1-p_m) - \beta_m p_m \\ \text{where} & \\ \alpha_m &= 0.1\frac{V_s+40}{1-e^{-(V_s+55)/10)}} \\ \beta_m &= 4e^{-(V_s+65)/18} \end{aligned} \tag{A.2}$$

We also have the sodium inactivation p_h given by:

$$\frac{\partial p_h}{\partial t} = \alpha_h(1 - p_h) - \beta_h p_h$$

where

$$\alpha_h = 0.07e^{-(V_s+65)/20} \tag{A.3}$$

$$\beta_h = \frac{1}{e^{-(V_s+35)/10} + 1}$$

B Activation Functions

Sigmoid Function

The logistic sigmoid function σ is given by:

$$\sigma(x) = \frac{1}{1+e^{-x}} \tag{B.1}$$

Where x is the input data. The sigmoid function maps input to a value between 0 and 1, useful when representing input as a probability. The output of the sigmoid function is essentially constant for changes in input as long as the input to the sigmoid function is either very positive or negative [34]. However, the output of the sigmoid function is susceptible to changes when the input values are close to zero due to the steep slope of σ around $x = 0$; see Figure 33.

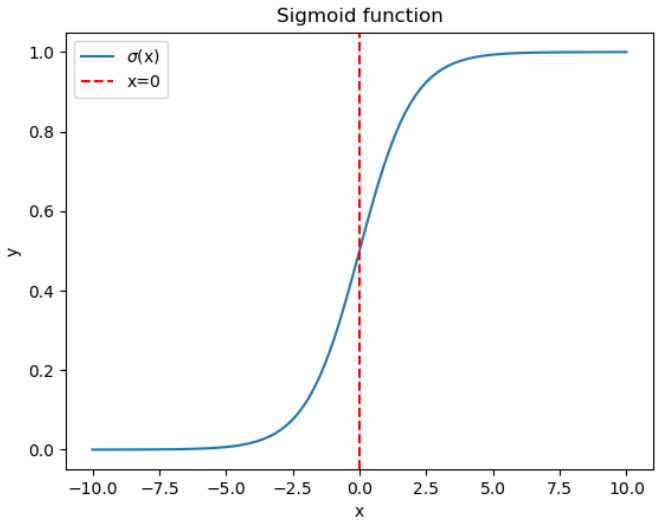

Figure 33: Plot of the sigmoid function $\sigma(x)|x \in [-10, 10]$ (see equation B.1). We can see that the slope of the sigmoid function is very steep around $x = 0$ and flat at higher and lower x values, which means that the output is sensitive to changes in input for values around 0 and insensitive to changes to larger absolute input values. Calculated and plotted by Sebastian Amundsen in Python.

ELU Function

The ELU activation function ELU(x) is given by:

$$\text{ELU}(x) = \begin{cases} x, & \text{if } x \geq 0 \\ \beta(e^x - 1), & \text{if } x < 0 \end{cases} \tag{B.2}$$

Where x is the input and $\beta = 1.0$. We can see that the ELU function is linear when $x \geq 0$ and increases minimally for $x < 0$; see Figure 34.

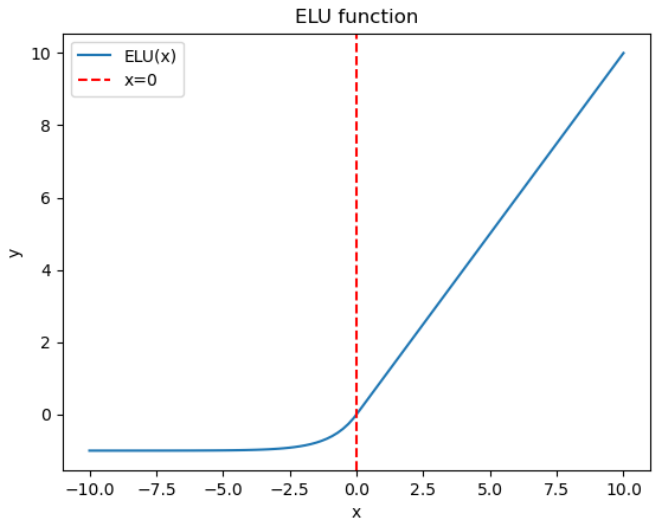

Figure 34: Plot of the ELU function ELU(x)$|x \in [-10, 10]$ (see equation B.2). The red line indicates the crossing of zero on the y-axis, and the blue line is ELU(x) as a function of the input. We can see a minimal increase in ELU(x) for input values lower than $x = 0$ and a linear increase in ELU(x) for input values larger than $x = 0$. Calculated and plotted by Sebastian Amundsen in Python.

C Code

The code for this project will eventually be publicly available through GitHub: https://github.com/Jonas-Verhellen/LFDeep. The code used for this project's main calculations is provided below. The first script loads the data before we run scripts two and three. After that, two and three calculate the different MTL models. The fourth script contains the LBTW task-balancing mechanism; the fifth is our program's entry file and calls our functions.

Listing 1: spike.py

```
import logging
import numpy as np
import pytorch_lightning as pl
import torch
from torch import nn
from torch.utils.data import Dataset, DataLoader, random_split
from typing import Optional

logger = logging.getLogger(__name__) # Used to track what is happening
    ↪ in the program, makes it much easier to debug the code.

class SpikeDataset(Dataset):
    def __init__(self,in_path,target_path):
        super().__init__()
        self.spike_data = np.load(in_path,mmap_mode='r+')
        self.target_data = np.load(target_path, mmap_mode='r+')[:,:,
            ↪ None]

    def __len__(self):
        return len(self.spike_data)

    def __getitem__(self, idx):
        data_sample = torch.from_numpy(self.spike_data[idx]).float()
        target_sample = torch.from_numpy(self.target_data[idx]).float()
        sample = {'data': data_sample, 'target': target_sample}
        return sample
```

```
25
26 class SpikeDataModule(pl.LightningDataModule): # This is what is used
       ↪ as input data
27     def __init__(self, config):
28         super().__init__()
29         self.prepare_data_per_node = True
30         self.in_path = config.in_path
31         self.target_path = config.target_path
32         self.batch_size = config.batch_size
33         self.num_workers = config.num_workers
34
35     def setup(self, stage: Optional[str] = None):
36         if stage == "fit" or stage is None:
37             train_set_full = SpikeDataset(self.in_path, self.
                ↪ target_path)
38             train_set_size = int(len(train_set_full) * 0.9)
39             valid_set_size = len(train_set_full) - train_set_size
40             self.train, self.validate = random_split(train_set_full, [
                ↪ train_set_size, valid_set_size])
41
42         if stage == "test" or stage is None:
43             self.test = SpikeDataset(self.in_path, self.target_path)
44
45     def train_dataloader(self):
46         return DataLoader(self.train, batch_size=self.batch_size,
            ↪ num_workers=self.num_workers)
47
48     def val_dataloader(self):
49         return DataLoader(self.validate, batch_size=self.batch_size,
            ↪ num_workers=self.num_workers)
50
51     def test_dataloader(self):
52         return DataLoader(self.test, batch_size=self.batch_size,
            ↪ num_workers=self.num_workers)
```

Listing 2: temporalconvolution.py

```
import torch
import torch.nn as nn
from torch.nn.utils import weight_norm
import torch.nn.functional as F
import pytorch_lightning as pl
torch.autograd.set_detect_anomaly(True)

class Chomp1d(pl.LightningModule):
    """Pad by (k-1)*d on the two sides of the input for convolution,
        ↪ and then use Chomp1d to remove the (k-1)*d elements on the
        ↪ right.
    This would essentially be the same as removing the "future elements
        ↪ ", which ensures causality. We are shifting the output of
        ↪ ordinary conv1d
    by (k-2)/2, where k is the kernel size."""

    def __init__(self, chomp_size):
        super(Chomp1d, self).__init__()
        self.chomp_size = chomp_size

    def forward(self, x):
        return x[:, :, :-self.chomp_size].contiguous()

class TemporalBlock(pl.LightningModule):
    """ Here we want to perform a weight normalization so that we are
        ↪ capable of performing stochastic gradient descent with
        ↪ respect to our weight parameters.
        Each of these temporal blocks consists of one net with two
            ↪ convolutional layers and
        one seperate convolutional layer which is defined as the
            ↪ downsample. """

    def __init__(self, n_inputs, n_outputs, kernel_size, stride,
        ↪ dilation, padding, dropout):
        super(TemporalBlock, self).__init__()
        self.conv1 = weight_norm(nn.Conv1d(n_inputs, n_outputs,
```

```
            ↪ kernel_size, stride=stride, padding=padding, dilation=
            ↪ dilation))
28          self.chomp1 = Chomp1d(padding)
29          self.sigmoid1 = nn.Sigmoid()
30          self.dropout1 = nn.Dropout(dropout)
31
32          self.conv2 = weight_norm(nn.Conv1d(n_outputs, n_outputs,
            ↪ kernel_size, stride=stride, padding=padding, dilation=
            ↪ dilation))
33          self.chomp2 = Chomp1d(padding)
34          self.sigmoid2 = nn.Sigmoid()
35          self.dropout2 = nn.Dropout(dropout)
36
37          self.net = nn.Sequential(self.conv1, self.chomp1, self.sigmoid1
            ↪ , self.dropout1, self.conv2, self.chomp2, self.sigmoid2,
            ↪  self.dropout2)
38          self.downsample = nn.Conv1d(n_inputs, n_outputs, 1) if n_inputs
            ↪  != n_outputs else None
39          self.sigmoid = nn.Sigmoid()
40          self.init_weights()
41
42      def init_weights(self):
43          """Initialize the weights according to a normal distribution.
44          Here we initialize the two convolutional layers and possibly a
            ↪ downsample convolutinal layer. """
45          self.conv1.weight.data.normal_(0, 0.01)
46          self.conv2.weight.data.normal_(0, 0.01)
47          if self.downsample is not None:
48              self.downsample.weight.data.normal_(0, 0.01)
49
50      def forward(self, x):
51          '''Here we pass through our network and compute our output. '''
52
53          out = self.net(x)
54          res = x if self.downsample is None else self.downsample(x) # If
            ↪  the input does not have the same number of elements as
            ↪ output --> downsample.
```

```
55          return self.sigmoid(out + res) # We add the input to the output
                ↪ .
56
57  class TemporalConvNet(pl.LightningModule):
58      '''Here we create our full temporal convolutional neural network. (
            ↪ Here the config file will include all the initilaization
            ↪ parameters).'''
59      def __init__(self, config):
60          super(TemporalConvNet, self).__init__()
61          self.num_inputs = config.num_inputs
62          self.num_channels = config.num_channels
63          self.kernel_size=config.kernel_size
64          self.dropout=config.dropout
65          layers = []
66          num_levels = len(self.num_channels)
67          for i in range(num_levels):
68              dilation_size = 2 ** i
69              in_channels = self.num_inputs if i == 0 else self.
                    ↪ num_channels[i-1] # For first block we use input
                    ↪ length.
70              out_channels = self.num_channels[i]
71              layers += [TemporalBlock(in_channels, out_channels, self.
                    ↪ kernel_size, stride=1, dilation=dilation_size,
                    ↪ padding=(self.kernel_size-1) * dilation_size,
                    ↪ dropout=self.dropout)]
72          self.network = nn.Sequential(*layers)
73
74      def forward(self, x):
75          '''Pass through the whole temporal convolutional net and
                ↪ generate an output. '''
76          out = self.network(x)
77          out = torch.flatten(out, start_dim=1) # Here we flatten the
                ↪ output from the convolutional neural network.
78          return out
```

Listing 3: mixtureofexperts.py

```
import hydra
import torch
import torch.nn as nn
import torch.nn.functional as F
import pytorch_lightning as pl
import torchmetrics
from omegaconf import OmegaConf
import numpy as np
from random import randint

class MH(pl.LightningModule):
    '''This is the Multi task Hard- parameter sharing model. For this model we only use one convolutional expert, defined as the shared bottom. This means that
    all of the tasks will use this shared bottom before the data is sent into the towers. '''
    def __init__(self, config):
        super(MH, self).__init__()
        self.num_tasks = config.num_tasks # This decides how many feed forward neural networks we are going to feed our data to .
        self.num_units = config.num_units # Number of neurons in the hidden layer of the towers.

        self.sequence_len = config.sequence_len # Length of input sequence.
        self.num_features = config.num_features

        self.optimizer = OmegaConf.load(hydra.utils.to_absolute_path(config.optimizer))

        self.shared_bottom = hydra.utils.instantiate(OmegaConf.load(hydra.utils.to_absolute_path(config.expert)))
        self.tcn_output_size = self.shared_bottom.num_channels[-1]*self.sequence_len # Produces tcn output size: 8 * 100

```

```
27          self.input_list = nn.ModuleList([nn.Linear(self.tcn_output_size
                ↪ , self.num_units) for _ in range(self.num_tasks)])
28          self.towers_list = nn.ModuleList(nn.Linear(self.num_units, self
                ↪ .num_units) for _ in range(self.num_tasks))
29          self.output_list = nn.ModuleList([nn.Linear(self.num_units, 1)
                ↪ for _ in range(self.num_tasks)])
30
31          # Now we want to define the metrics, which are used to evaluate
                ↪  the performance of our model:
32          self.loss_fn = nn.MSELoss()
33
34          self.loss_fn_spikes = nn.BCEWithLogitsLoss()
35
36          self.training_metric =  torchmetrics.MeanSquaredError()
37          self.training_metric_spike = torchmetrics.Accuracy()#task='
                ↪ binary')
38          self.validation_metric = torchmetrics.MeanSquaredError()
39          self.validation_metric_spike = torchmetrics.Accuracy()#task='
                ↪ binary')
40          self.test_metric =  torchmetrics.MeanSquaredError()
41          self.test_metric_spike = torchmetrics.Accuracy()#task='binary')
42
43          # Here we add an optional balancing method which we use the
                ↪ adjust the losses of the different tasks.
44          self.balancer = hydra.utils.instantiate(OmegaConf.load(hydra.
                ↪ utils.to_absolute_path(config.balancer)))
45
46      def balanced_loss_function(self, predictions, predictions_spike,
            ↪ targets, targets_spike):
47          """Here we want to create our own loss function which should
                ↪ calculate the loss for each compartment
48          I want to incorporate the MSE loss function. Here we will also
                ↪ be adding a balancer method (LBTW).
49          (The dimensions of the predictions tensor is (batch_size,
                ↪ num_tasks)"""
50
51          lamb = torch.ones(self.num_tasks) # This is the initial lambda
```

```
        ↪ array.
52
53          task_losses = torch.zeros(self.num_tasks) # Here we will store
                ↪ our task loss values.
54          with torch.no_grad():
55              for batch in range(predictions.shape[0]):
56                  loss_spike = self.loss_fn_spikes(predictions_spike[
                        ↪ batch], targets_spike[batch])
57                  task_losses[-1] = loss_spike * lamb[-1]
58                  self.balancer.get_initial_loss(task_losses[-1], self.
                        ↪ num_tasks-1)
59                  for task in range(self.num_tasks-1):
60                      loss = self.loss_fn(predictions[batch, task],
                            ↪ targets[batch, task])
61                      task_losses[task] = loss * lamb[task] # Have to
                            ↪ calculate the loss for each task.
62
63                      if batch == 0: # First batch:
64                          self.balancer.get_initial_loss(task_losses[task
                                ↪ ], task)
65
66                      self.balancer.LBTW(task_losses[task], task)
67
68
69                  self.balancer.LBTW(task_losses[-1], self.num_tasks-1)
70
71                  weights = torch.Tensor(self.balancer.get_weights())
72
73                  lamb = weights
74
75          if (task_losses != task_losses).any():
76              raise ValueError("Loss contains NaN values")
77          if torch.isinf(task_losses).any():
78              raise ValueError("Loss contains infinite values")
79
80          weights = weights.to(device="cuda")
81          task_losses = task_losses.to(device="cuda")
```

```
82          task_losses.requires_grad=True
83          mse_loss = torch.mean( nn.MSELoss(reduce = False)(predictions,
                ↪ targets), axis=0 ) # Mean over all the batches.
84
85          total_loss = ( mse_loss@weights[:-1] / len(weights[:-1]) ) +
                ↪ self.loss_fn_spikes(predictions_spike,targets_spike) *
                ↪ weights[-1]
86          total_loss = total_loss.to(device="cuda")
87
88          return total_loss
89
90      def forward(self, inputs, diversity = False):
91          """ This is the function were we generate our output from all
                ↪ the different tasks. """
92          shared_bottom_outputs = self.calculating_shared_bottom(inputs)
93          if torch.sum(torch.isnan(shared_bottom_outputs)):
94              print('found nanz')
95          output = []
96          for task in range(self.num_tasks):
97              aux = self.input_list[task](shared_bottom_outputs)
98              aux = F.elu(aux)
99              aux = self.towers_list[task](aux)
100             aux = F.elu(aux)
101             aux = self.output_list[task](aux)
102             output.append(aux) # Here we append the output
                    ↪ corresponding to each specific task.
103
104         output = torch.cat([x.float() for x in output], dim=1) # Links
                ↪ togheter the given sequence tensors in the given
                ↪ dimension.
105         return output
106
107     def calculating_shared_bottom(self, inputs):
108         """ Calculating the shared bottom, where the activation
                ↪ function is ReLU."""
109
110         aux = self.shared_bottom(inputs) # Here we collect the list
```

```
            ↪ consisting of the shared bottom kernel.
111         shared_bottom_outputs = F.relu(aux,inplace=False) # Perform the
            ↪  relu activation function on the reshaped output.
112
113         return shared_bottom_outputs
114
115     def compute_element_errors(self, pred_out, true_out):
116         return torch.mean((pred_out-true_out)**2,0)
117
118     def training_step(self, batch, batch_idx):
119         data, targets = batch['data'], batch['target']
120         predictions = self(data)
121         predictions_spike, targets_spike = predictions[:,639], targets
            ↪ [:,639,-1]
122         predictions, targets = torch.cat([predictions[:,:639],
            ↪ predictions[:,640,None]],1), torch.cat([targets
            ↪ [:,:639,-1],targets[:,640,-1,None]],1)
123         loss = self.balanced_loss_function(predictions,
            ↪ predictions_spike, targets, targets_spike) # Here we
            ↪ gather the balanced loss for each compartment.
124         return {'loss': loss, 'predictions': predictions, 'targets':
            ↪ targets, 'predictions_spike': predictions_spike, '
            ↪ targets_spike': targets_spike}
125
126     def training_step_end(self, outputs):
127         """ This is called after the training_step method has been
            ↪ called for all batches in the current epoch. We
128         use it for logging before we move on to the next epoch."""
129         self.training_metric(outputs['predictions'], outputs['targets'
            ↪ ])
130         self.training_metric_spike(F.softmax(outputs['predictions_spike
            ↪ ']).int(),outputs['targets_spike'].int())
131         self.log('loss/train', outputs['loss'])
132         self.log('metric/train', self.training_metric)
133         self.log('metric/train/spike', self.training_metric_spike)
134
135     def validation_step(self, batch, batch_idx):
```

```
136         data, targets = batch['data'], batch['target']
137         with torch.no_grad():
138             predictions = self(data)
139             predictions_spike, targets_spike = predictions[:,639],
                ↪ targets[:,639,-1]
140             predictions, targets = torch.cat([predictions[:,:639],
                ↪ predictions[:,640,None]],1), torch.cat([targets
                ↪ [:,:639,-1],targets[:,640,-1,None]],1)
141             loss = self.loss_fn(predictions, targets) + self.
                ↪ loss_fn_spikes(predictions_spike,targets_spike)
142         return {'loss': loss, 'predictions': predictions, 'targets':
            ↪ targets, 'predictions_spike': predictions_spike, '
            ↪ targets_spike': targets_spike}
143
144     def validation_step_end(self, outputs):
145         element_errors = self.compute_element_errors(outputs['
            ↪ predictions'],outputs['targets'])
146         self.validation_metric(outputs['predictions'], outputs['targets
            ↪ '])
147         self.validation_metric_spike(F.softmax(outputs['
            ↪ predictions_spike']).int(),outputs['targets_spike'].int
            ↪ ())
148         self.log('loss/val', outputs['loss'])
149         self.log('metric/val', self.validation_metric)
150         self.log('metric/val/spike',self.validation_metric_spike)
151         for i in range(len(element_errors)):
152             self.log('metric/val/element_errors_'+str(i),element_errors
                ↪ [i])
153
154     def test_step(self, batch, batch_idx):
155         data, targets = batch['data'], batch['target']
156
157         with torch.no_grad():
158             predictions = self(data)
159             predictions_spike, targets_spike = predictions[:,639],
                ↪ targets[:,639,-1]
160             predictions, targets = torch.cat([predictions[:,:639],
```

```
                ↪ predictions[:,640,None]],1), torch.cat([targets
                ↪ [:,:639,-1],targets[:,640,-1,None]],1)
161             loss = self.loss_fn(predictions, targets) + self.
                ↪ loss_fn_spikes(predictions_spike,targets_spike)
162         return {'loss': loss, 'predictions': predictions, 'targets':
                ↪ targets, 'predictions_spike': predictions_spike, '
                ↪ targets_spike': targets_spike}
163
164     def test_step_end(self, outputs):
165         self.test_metric(outputs['predictions'], outputs['targets'])
166         self.validation_metric_spike(F.softmax(outputs['
                ↪ predictions_spike']).int(),outputs['targets_spike'].int
                ↪ ())
167         self.log('loss/test', outputs['loss'])
168         self.log('metric/test', self.test_metric)
169         self.log('metric/test/spike',self.test_metric_spike)
170
171     def configure_optimizers(self):
172         return hydra.utils.instantiate(self.optimizer, self.parameters
                ↪ ())
173
174     def on_train_start(self):
175         self.logger.log_hyperparams(self.hparams, {"metric/training":
                ↪ 0, "metric/test": 0, "metric/val": 0})
176
177 class MMoE(pl.LightningModule):
178     def __init__(self, config):
179         super(MMoE, self).__init__()
180         self.save_hyperparameters()
181         self.num_tasks = config.num_tasks
182         self.num_experts = config.num_experts
183         self.num_units = config.num_units
184
185         self.sequence_len = config.sequence_len
186         self.num_features =  config.num_features
187
188         self.use_expert_bias = config.use_expert_bias
```

```
189         self.use_gate_bias = config.use_gate_bias
190         self.optimizer = OmegaConf.load(hydra.utils.to_absolute_path(
                ↪ config.optimizer))
191 
192         self.expert_kernels_tcn = nn.ModuleList([hydra.utils.
                ↪ instantiate(OmegaConf.load(hydra.utils.to_absolute_path(
                ↪ config.expert))) for _ in range(self.num_experts)])
193         self.tcn_output_size = self.expert_kernels_tcn[0].num_channels
                ↪ [-1]*self.sequence_len
194         self.compressor = hydra.utils.instantiate(OmegaConf.load(hydra.
                ↪ utils.to_absolute_path(config.expert)))
195 
196         self.input_list = nn.ModuleList([nn.Linear(self.tcn_output_size
                ↪ , self.num_units) for _ in range(self.num_tasks)])
197         self.towers_list = nn.ModuleList(nn.Linear(self.num_units, self
                ↪ .num_units) for _ in range(self.num_tasks))
198         self.output_list = nn.ModuleList([nn.Linear(self.num_units, 1)
                ↪ for _ in range(self.num_tasks)])
199 
200         if self.use_expert_bias:
201             """ Here we set a bias parameter for the experts that
                    ↪ pytoch lightning keeps track of and updates."""
202             self.expert_bias = nn.Parameter(torch.zeros(self.
                    ↪ num_experts, self.tcn_output_size), requires_grad=
                    ↪ True)
203 
204         gate_kernels = torch.rand((self.num_tasks, self.tcn_output_size
                ↪ , self.num_experts)).float()
205         self.gate_kernels = nn.Parameter(gate_kernels, requires_grad=
                ↪ True)
206 
207         if self.use_gate_bias:
208             """ Set bias for the gates"""
209             self.gate_bias = nn.Parameter(torch.zeros(self.num_tasks,
                    ↪ 1, self.num_experts), requires_grad=True)
210 
211         self.task_bias = nn.Parameter(torch.zeros(self.num_tasks),
```

```
            ↪ requires_grad=True) # Set task biases.
212
213         self.loss_fn = nn.MSELoss()
214         self.loss_fn_spikes = nn.BCEWithLogitsLoss()
215
216         self.training_metric =  torchmetrics.MeanSquaredError()
217         self.training_metric_spike = torchmetrics.Accuracy()#task='
            ↪ binary')
218         self.validation_metric = torchmetrics.MeanSquaredError()
219         self.validation_metric_spike = torchmetrics.Accuracy()#task='
            ↪ binary')
220         self.test_metric =  torchmetrics.MeanSquaredError()
221         self.test_metric_spike = torchmetrics.Accuracy()#task='binary')
222
223         # Here we add an optional balancing method which we use the
            ↪ adjust the losses of the different tasks.
224         self.balancer = hydra.utils.instantiate(OmegaConf.load(hydra.
            ↪ utils.to_absolute_path(config.balancer)))
225
226     def balanced_loss_function(self, predictions, predictions_spike,
        ↪ targets, targets_spike):
227         """Here we want to create our own loss function which should
            ↪ calculate the loss for each compartment
228         I want to incorporate the MSE loss function. Here we will also
            ↪ be adding a balancer method (LBTW).
229         (The dimensions of the predictions tensor is (batch_size,
            ↪ num_tasks)"""
230
231         lamb = torch.ones(self.num_tasks) # This is the initial lambda
            ↪ array.
232
233         task_losses = torch.zeros(self.num_tasks) # Here we will store
            ↪ our task loss values.
234         with torch.no_grad():
235             for batch in range(predictions.shape[0]):
236                 loss_spike = self.loss_fn_spikes(predictions_spike[
                    ↪ batch], targets_spike[batch])
```

```
237                 task_losses[-1] = loss_spike * lamb[-1]
238                 self.balancer.get_initial_loss(task_losses[-1], self.
                        ↪ num_tasks-1)
239                 for task in range(self.num_tasks-1):
240                     loss = self.loss_fn(predictions[batch, task],
                            ↪ targets[batch, task])
241                     task_losses[task] = loss * lamb[task] # Have to
                            ↪ calculate the loss for each task.
242
243                     if batch == 0: # First batch:
244                         self.balancer.get_initial_loss(task_losses[task
                                ↪ ], task)
245                     self.balancer.LBTW(task_losses[task], task)
246                 self.balancer.LBTW(task_losses[-1], self.num_tasks-1)
247
248                 weights = torch.Tensor(self.balancer.get_weights())
249
250                 lamb = weights
251
252         if (task_losses != task_losses).any():
253             raise ValueError("Loss contains NaN values")
254         if torch.isinf(task_losses).any():
255             raise ValueError("Loss contains infinite values")
256
257         weights = weights.to(device="cuda")
258         task_losses = task_losses.to(device="cuda")
259         task_losses.requires_grad=True
260         mse_loss = torch.mean( nn.MSELoss(reduce = False)(predictions,
                ↪ targets), axis=0 )
261
262         total_loss = ( mse_loss@weights[:-1] / len(weights[:-1]) ) +
                ↪ self.loss_fn_spikes(predictions_spike,targets_spike) *
                ↪ weights[-1]
263         total_loss = total_loss.to(device="cuda")
264
265         return total_loss
266
```

```
267     def forward(self, inputs, diversity=False):
268         batch_size = inputs.shape[0]
269         expert_outputs = self.calculating_experts(inputs)
270         gate_outputs = self.calculating_gates(inputs, batch_size)
271         product_outputs = self.multiplying_gates_and_experts(
                ↪ expert_outputs, gate_outputs)
272 
273         output = []
274         for task in range(self.num_tasks):
275             aux = self.input_list[task](product_outputs[task,:,:])
276             aux = F.elu(aux)
277             aux = self.towers_list[task](aux)
278             aux = F.elu(aux)
279             aux = self.output_list[task](aux)
280             output.append(aux)
281 
282         output = torch.cat([x.float() for x in output], dim=1)
283         if diversity:
284             return output, expert_outputs
285         else:
286             return output
287 
288     def calculating_experts(self, inputs):
289         """
290         Calculating the experts
291         """
292 
293         for i in range(self.num_experts):
294             aux = self.expert_kernels_tcn[i](inputs)
295             if i == 0:
296                 expert_outputs = aux.reshape(1, aux.shape[0], aux.shape
                        ↪ [1])
297             else:
298                 expert_outputs = torch.cat((expert_outputs, aux.reshape
                        ↪ (1, aux.shape[0], aux.shape[1])), dim=0)
299 
300         if self.use_expert_bias:
```

```
301             for expert in range(self.num_experts):
302                     expert_bias = self.expert_bias[expert]
303                     expert_outputs[expert] = expert_outputs[expert].add
                        ↪ (expert_bias[None, :])
304         expert_outputs = F.relu(expert_outputs,inplace=False)
305         return expert_outputs
306 
307     def calculating_gates(self, inputs, batch_size):
308         """
309         Calculating the gates, g^{k}(x) = activation(W_{gk} * x + b),
                ↪ where activation is softmax according to the paper T x n
                ↪  x E.
310         gate outputs are found by doing a matrix multiplication between
                ↪  the compressed inputs and the gate kernels
311         for index = 0 and between gate outputs and compressed inputs
                ↪ for the remaining indicies.
312         """
313         compressed_inputs = self.compressor(inputs)
314 
315         for index in range(self.num_tasks):
316             if index == 0:
317                 gate_outputs = torch.mm(compressed_inputs, self.
                        ↪ gate_kernels[index]).reshape(1, batch_size, self
                        ↪ .num_experts)
318             else:
319                 gate_outputs = torch.cat((gate_outputs, torch.mm(
                        ↪ compressed_inputs, self.gate_kernels[index]).
                        ↪ reshape(1, batch_size, self.num_experts)), dim
                        ↪ =0)
320 
321         if self.use_gate_bias:
322             gate_outputs = gate_outputs.add(self.gate_bias)
323 
324         gate_outputs = F.softmax(gate_outputs, dim=2) # Dim=2 -->
                ↪ Normalizes values along axis 2.
325         return gate_outputs
326 
```

```
327     def multiplying_gates_and_experts(self, expert_outputs,
            ↪ gate_outputs):
328         """
329         Multiplying gates and experts
330         """
331 
332         for task in range(self.num_tasks):
333             gate = gate_outputs[task]
334             for expert in range(self.num_experts):
335                 gate_output = gate[:, expert]
336                 product = expert_outputs[expert] * gate_output[:, None]
337                 if expert == 0:
338                     products = product
339                 else:
340                     products = products.add(product)
341             final_product = products.add(self.task_bias[task])
342 
343             if task == 0:
344                 final_products = final_product.reshape(1, final_product
                        ↪ .shape[0], final_product.shape[1])
345             else:
346                 final_products = torch.cat((final_products,
                        ↪ final_product.reshape(1, final_product.shape[0],
                        ↪  final_product.shape[1])), dim=0)
347         return final_products
348 
349     def compute_diversity(self, batch):
350         import project.utils.diversity_metrics as dm
351         batch = torch.reshape(batch,[batch.shape[0],batch.shape[1]*
                ↪ batch.shape[2]])
352         diversity_matrix = dm.diversity_matrix(batch.T)
353         diversity_score = torch.mean(diversity_matrix)
354         diversity_determinant = torch.linalg.det(diversity_matrix)
355         diversity_permanent = dm.permanent(diversity_matrix)
356         return diversity_score, diversity_determinant,
                ↪ diversity_permanent
357 
```

```
358     def compute_element_errors(self, pred_out, true_out):
359         return torch.mean((pred_out-true_out)**2,0)
360 
361     def training_step(self, batch, batch_idx):
362         data, targets = batch['data'], batch['target']
363         predictions = self(data)
364         predictions_spike, targets_spike = predictions[:,639], targets
                ↪ [:,639,-1]
365         predictions, targets = torch.cat([predictions[:,:639],
                ↪ predictions[:,640,None]],1), torch.cat([targets
                ↪ [:,:639,-1],targets[:,640,-1,None]],1)
366         loss = self.balanced_loss_function(predictions,
                ↪ predictions_spike, targets, targets_spike) # Here we
                ↪ gather the balanced loss for each compartment.
367         return {'loss': loss, 'predictions': predictions, 'targets':
                ↪ targets, 'predictions_spike': predictions_spike, '
                ↪ targets_spike': targets_spike}
368 
369     def training_step_end(self, outputs):
370         self.training_metric(outputs['predictions'], outputs['targets'
                ↪ ])
371         self.training_metric_spike(F.softmax(outputs['predictions_spike
                ↪ ']).int(),outputs['targets_spike'].int())
372         self.log('loss/train', outputs['loss'])
373         self.log('metric/train', self.training_metric)
374         self.log('metric/train/spike', self.training_metric_spike)
375 
376     def validation_step(self, batch, batch_idx):
377         data, targets = batch['data'], batch['target']
378         with torch.no_grad():
379             predictions = self(data)
380             predictions_spike, targets_spike = predictions[:,639],
                    ↪ targets[:,639,-1]
381             predictions, targets = torch.cat([predictions[:,:639],
                    ↪ predictions[:,640,None]],1), torch.cat([targets
                    ↪ [:,:639,-1],targets[:,640,-1,None]],1)
382             loss = self.loss_fn(predictions, targets) + self.
```

```
                ↪ loss_fn_spikes(predictions_spike,targets_spike)
383         return {'loss': loss, 'predictions': predictions, 'targets':
                ↪ targets, 'predictions_spike': predictions_spike, '
                ↪ targets_spike': targets_spike, 'current_batch': data}
384
385     def validation_step_end(self, outputs):
386         expert_output = self.calculating_experts(outputs['current_batch
                ↪ '])
387         element_errors = self.compute_element_errors(outputs['
                ↪ predictions'],outputs['targets'])
388         diversity_score, diversity_determinant, diversity_permanent =
                ↪ self.compute_diversity(expert_output)
389         self.validation_metric(outputs['predictions'], outputs['targets
                ↪ '])
390         self.validation_metric_spike(F.softmax(outputs['
                ↪ predictions_spike']).int(),outputs['targets_spike'].int
                ↪ ())
391         self.log('loss/val', outputs['loss'])
392         self.log('metric/val', self.validation_metric)
393         self.log('metric/val/spike',self.validation_metric_spike)
394         for i in range(len(element_errors)):
395             self.log('metric/val/element_errors_'+str(i),element_errors
                    ↪ [i])
396         self.log('diversity/val/score', diversity_score)
397         self.log('diversity/val/determinant',diversity_determinant)
398         self.log('diversity/val/permanent',diversity_permanent)
399
400     def test_step(self, batch, batch_idx):
401         data, targets = batch['data'], batch['target']
402
403         with torch.no_grad():
404             predictions = self(data)
405             predictions_spike, targets_spike = predictions[:,639],
                    ↪ targets[:,639,-1]
406             predictions, targets = torch.cat([predictions[:,:639],
                    ↪ predictions[:,640,None]],1), torch.cat([targets
                    ↪ [:,:639,-1],targets[:,640,-1,None]],1)
```

```
407             loss = self.loss_fn(predictions, targets) + self.
                    ↪ loss_fn_spikes(predictions_spike,targets_spike)
408         return {'loss': loss, 'predictions': predictions, 'targets':
                ↪ targets, 'predictions_spike': predictions_spike, '
                ↪ targets_spike': targets_spike}
409
410     def test_step_end(self, outputs):
411         self.test_metric(outputs['predictions'], outputs['targets'])
412         self.test_metric_spike(F.softmax(outputs['predictions_spike']).
                ↪ int(),outputs['targets_spike'].int())
413         self.log('loss/test', outputs['loss'])
414         self.log('metric/test', self.test_metric)
415         self.log('metric/test/spike',self.test_metric_spike)
416
417     def configure_optimizers(self):
418         return hydra.utils.instantiate(self.optimizer, self.parameters
                ↪ ())
419
420     def on_train_start(self):
421         self.logger.log_hyperparams(self.hparams, {"metric/training":
                ↪ 0, "metric/test": 0, "metric/val": 0})
422
423 class MMoEEx(MMoE):
424     def __init__(self, config):
425         super(MMoEEx, self).__init__(config)
426         self.prob_exclusivity = config.prob_exclusivity
427         self.type = config.type
428
429         exclusivity = np.repeat(self.num_tasks + 1, self.num_experts)
430         to_add = int(self.num_experts * self.prob_exclusivity)
431         for e in range(to_add):
432             exclusivity[e] = randint(0, self.num_tasks)
433
434         self.exclusivity = exclusivity
435         gate_kernels = torch.rand((self.num_tasks, self.tcn_output_size
                ↪ , self.num_experts)).float()
436
```

```
437             for expert_number, task_number in enumerate(self.exclusivity):
438                 if task_number < self.num_tasks + 1:
439                     if self.type == "exclusivity":
440                         for task in range(self.num_tasks):
441                             if task != task_number:
442                                 gate_kernels[task][:, expert_number] = 0.0
443                     else:
444                         gate_kernels[task_number][:, expert_number] = 0.0
445 
446             self.gate_kernels = nn.Parameter(gate_kernels, requires_grad=
                    ↪ True)
```

Listing 4: taskbalancing.py

```
"""
Copyright (c) 2020-present, Royal Bank of Canada.
All rights reserved.
 This source code is licensed under the license found in the
 LICENSE file in the root directory of this source tree.
Task balacing approaches for multi-task learning
Two methods based on loss ratio called:
- DWA - Dynamic Weight Average
- LBTW - Loss Balanced Task Weighting
Written by Gabriel Oliveira in pytorch
"""
import torch
import torch.nn as nn
import torch.nn.functional as F
from numpy.random import randint, binomial
import numpy as np
import hydra
from omegaconf import OmegaConf
from omegaconf import DictConfig, OmegaConf

class TaskBalanceMTL:
    # Class for task balancing methods
    def __init__(self, config):
        # Hyper parameters
        self.balance_method = config.balance_method
        self.K = config.n_tasks
        self.T = config.n_tasks
        self.alpha_balance = config.alpha_balance
        self.n_tasks = config.n_tasks
        self.task_ratios = torch.zeros([self.n_tasks])
        self.task_weights = torch.zeros([self.n_tasks])
        self.initial_losses = torch.zeros([self.n_tasks])
        self.weight_history = []
        self.history_last = []
        for i in range(self.n_tasks):
            self.weight_history.append([])
```

```
37              self.history_last.append([])
38
39          # Setting weight method
40          self.balance_mode = config.balance
41          if self.balance_mode == "DWA":
42              print("...DWA Weight balance")
43          if self.balance_mode == "LBTW":
44              print("...LBTW Weight balance")
45
46      def add_loss_history(self, task_losses):
47          for i in range(0, self.n_tasks):
48              self.weight_history[i].append(task_losses[i])
49
50      def last_elements_history(self):
51          for i in range(0, self.n_tasks):
52              self.history_last[i] = self.weight_history[i][-2:]
53
54      def compute_ratios(self, task_losses, epoch):
55
56          for i in range(0, self.n_tasks):
57              if epoch <= 1:
58                  self.task_ratios[:] = 1
59              else:
60                  before = "-"
61                  if self.history_last[i][-2] > -0.01 and self.
                        ↪ history_last[i][-2] < 0.01:
62                      before = self.history_last[i][-2]
63                      self.history_last[i][-2] = 0.01
64
65                  self.task_ratios[i] = (
66                      self.history_last[i][-1] / self.history_last[i][-2]
67                  )
68
69      def sum_losses_tasks(self):
70          ratios_sum = 0.0
71          for i in range(0, self.n_tasks):
72              ratios_sum += torch.exp(self.task_ratios[i] / self.T)
```

```
73          return ratios_sum
74
75      def DWA(self, task_losses, epoch):
76          self.compute_ratios(task_losses, epoch)
77          ratios_sum = self.sum_losses_tasks()
78
79          for i in range(0, self.n_tasks):
80              self.task_weights[i] = max(
81                  min((self.K * torch.exp(self.task_ratios[i] / self.T))
                        ↪ / ratios_sum, 1.5),
82                  0.5,
83              )
84
85      def get_weights(self):
86          return self.task_weights
87
88      def get_initial_loss(self, losses, task):
89          self.initial_losses[task] = losses
90
91      def LBTW(self, batch_losses, task):
92          self.task_weights[task] = torch.fmax(
93              torch.fmin(pow(batch_losses / self.initial_losses[task],
                    ↪ torch.Tensor([self.alpha_balance])), torch.Tensor
                    ↪ ([1.0])),
94              torch.Tensor([0.01]),
95          )
```

Listing 5: main.py

```python
import hydra
import logging
import pytorch_lightning as pl
from omegaconf import DictConfig, OmegaConf
logger = logging.getLogger(__name__)
import torch
torch.cuda.empty_cache()

@hydra.main(config_path="configs", config_name="defaults")
def main(config: DictConfig) -> None:
    torch.cuda.empty_cache()
    pl.seed_everything(2509)
    logger.info("\n" + OmegaConf.to_yaml(config))

    # Instantiate all modules specified in the configs
    model = hydra.utils.instantiate(config.model)
    data_module = hydra.utils.instantiate(config.data)

    # Let hydra manage direcotry outputs
    tensorboard = pl.loggers.TensorBoardLogger(".", "", "", log_graph=
        ↪ True, default_hp_metric=False)
    checkpoint_callback = pl.callbacks.ModelCheckpoint(save_top_k=2,
        ↪ monitor="loss/val", mode="min", filename="sample-mh-{epoch
        ↪ :02d}-{val_loss:.2f}",)
    early_stopping_callback = pl.callbacks.EarlyStopping(monitor='loss/
        ↪ val', patience=500)
    callbacks = [checkpoint_callback, early_stopping_callback]

    trainer = pl.Trainer(**OmegaConf.to_container(config.trainer),
        ↪ logger=tensorboard, callbacks=callbacks, auto_lr_find=True,
        ↪ precision=16, accelerator='gpu', devices=4, strategy="ddp")
    trainer.fit(model, datamodule=data_module) # This method calls on
        ↪ the forward function and uses dataloader as input.
    trainer.test(model, datamodule=data_module)  # Optional


```

```
30 if __name__ == '__main__':
31     main()
```