Training deep learning models to classify input to simulations of a biological neural network

by

Samuel Korsan Knudsen

Thesis

for the degree of

Master of Science

Faculty of Mathematics and Natural Sciences
University of Oslo

November 2019

Abstract

We present a computationally effective toy model of the visual system of a biological brain, that can easily be extended to add more realism. The model takes images as input – representing visual stimuli from the eye – and outputs an estimation of the cortical LFP (local field potential) that is generated as cortex processes the input. We run a large number of simulations, each stimulated by a randomized sequence of 10 images, and use the output data to train deep learning algorithms (CNN and LSTM) to classify pieces of the LFP by input image.

The classifiers reach accuracies of 66 and 65%, averaged across all 10 inputs, suggesting that the LFP indeed contain information about the stimulus that a brain is processing. They are also more likely to confuse the LFPs of images that qualitatively seem visually similar. We observe that a trained CNN transfers better to test data that deviates slightly from the training set, but that the LSTM seems marginally better at handling noise.

Sammendrag

Vi presenterer en beregningseffektiv, forenklet modell av synssystemet i en biologisk hjerne, som enkelt kan utvides til å inkludere mer realisme. Modellen simulerer visuell stimuli fra øyet ved å ta bilder som input, og beregner så det lokale elektriske signalet (LFP) som oppstår i hjernen når synssenteret prosesserer informasjonen. Vi kjører et stort antall simuleringer, hvor hver tar en sekvens med 10 bilder i tilfeldig rekkefølge som input, og bruker den produserte dataen til å trene opp kunstig intelligens (AI) til å klassifisere LFP-signalene etter input-bilde.

Til dette bruker vi to populære maskinlæringsalgoritmer, kjent som CNN og LSTM. Algoritmene oppnår presisjonsverdier på henholdsvis 66 og 65%, gjennomsnittlig over alle de 10 bildene, noe som tyder på at det elektriske feltet absolutt inneholder informasjon om stimulien som en hjerne prosesserer. De er også mer tilbøyelig til å blande LFP-er fra bilder som kvalitativt ligner visuelt. Vi observerer at et trent CNN gjør det bedre på data som skiller seg litt fra treningsdataen, men at et LSTM ser ut til å håndtere støy noe bedre.

Acknowledgements

Thanks to all my friends and family for all support and for letting me be the absentminded troglodyte that I am. Thanks to everyone at CINPLA for creating a fantastic work environment and for providing valuable input when needed. A special thanks to my supervisors Alex, Espen and Gaute – it really has been a privilege to get to do this work, and I could of course not have done it without you guys.

Shoutout to my people and colleagues at CompPhys and Corporated Inc, and to Gunnar for bequeathing his vinyl collection to me in trade of a special mention here. Lastly a big up to Lånekassen – you, you're good, you!

Contents

Chapter 1

Introduction

In this master thesis we study the brain – the tool with which the universe observes itself. Even though it might be the most complex thing that we know of, we are gradually understanding more and more about how it works. Ever since Luigi Galvani experimented on frogs in the 1700s we have known about its electrical nature [30], and from Alan Hodkin and Andrew Huxley's Nobel Price winning work in the 1950s [19] we have begun to grasp the mathematics behind it.

The field of brain research, known as **neuroscience**, is a junction where many different fields of science meet and cooperate. Physics, biology, chemistry, psychology and many others are all applied in various ways to do research in the field. The past decades' advances in computational technology have spawned the subfield **computational neuroscience**, a field that promises further theoretical understanding of the brain by letting neuroscientists incorporate simulations and mathematical modelling in their work.

Findings in neuroscience have given rise to the field of deep learning, an area of **artificial intelligence** (AI) that attempts to mimic the brain's way of processing information in computer programs. This is another field that has benefited greatly from increased access to computing power, and AI algorithms such as artificial neural networks (ANNs) have now surpassed human level skill in many areas, like reading facial cues [24] and playing Chess [35].

The topic of this thesis involves both AI and computational neuroscience. Here we build a simulation of an experiment that scientists have performed using biological brains [13]. More specifically, we simulate a very simplified model of the part of the brain involved with vision, known as the **visual system**. The model represents an eye that forwards optical information into the visual cortex, via the lateral geniculate nucleus (LGN).

The simulation then calculates the electric field, known as the the **LFP** (local field potential), that results from the electrical activity in cortex as the

information is processed. Finally we will feed this signal into an AI to see if it can predict the visual stimulus using only the information found in the LFP. An illustration of the whole experiment is seen in figure 1.1.

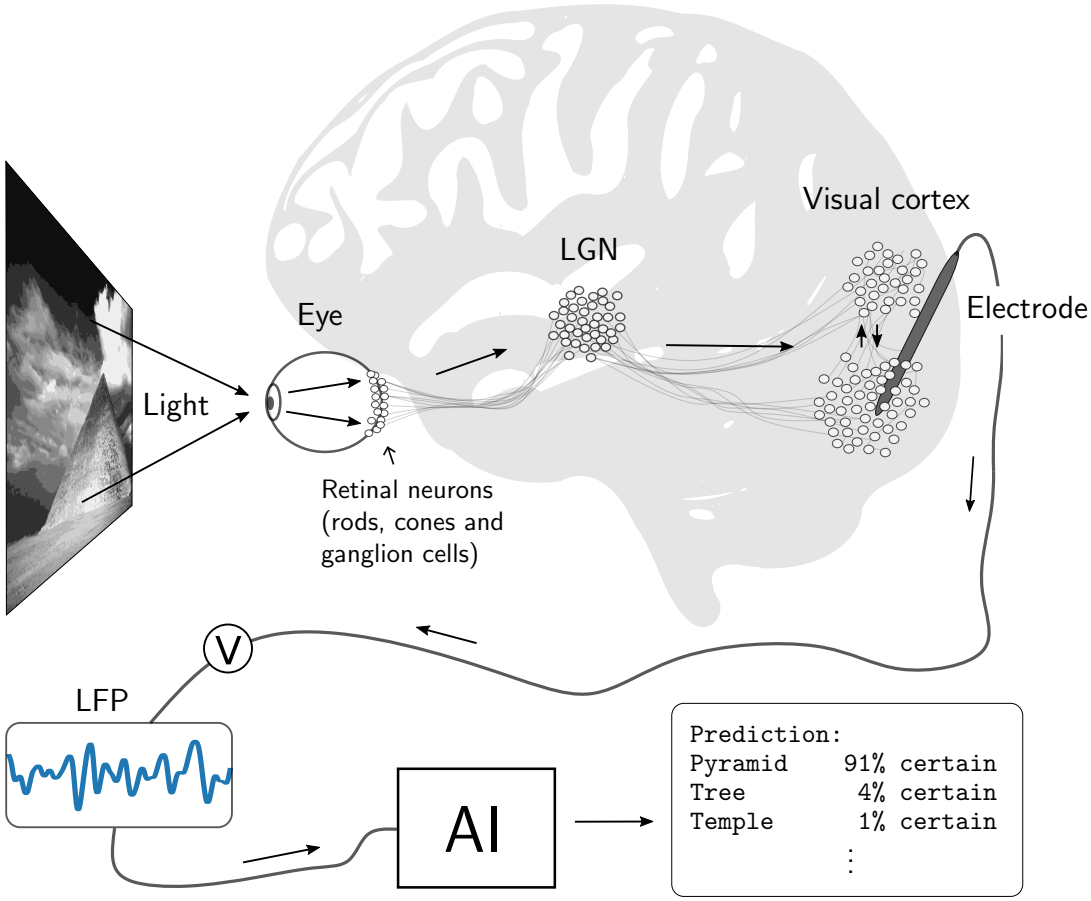


Figure 1.1: An illustration of the experiment. We simulate the visual system of a brain that is looking at an image. The optical information is filtered through a part of the brain called the LGN (Lateral Geniculate Nucleus) before it is sent to the visual cortex. An artificial intelligence then reads the recording of the LFP in cortex, and tries to predict what the image was. The clusters of circles represent distinct populations of neurons, and the arrows represent the direction of propagation of the information.

1.1 Motivation

The use of mathematical modeling in combination with experiments is one of humanity's most powerful inventions. It is what allowed the field of physics to become the powerhouse of science that it became during the 19th and 20th century, which directly lead to the existence of most of the amazing technology around us today. Mathematical modeling in biology have for a long time been an almost impossible endeavour due to the sheer size and complexity of biological systems, but with modern computing powers those days have come to an end. To paraphrase the distinguished physicist Geoffrey West, author of Scale [41, 40]:

> Mathematized physics was the science of the 19th and 20th century. Mathematized biology will be the science of the 21st century.

Any scientific theory is built on a set of assumptions and ideas about mechanisms, and by defining those in mathematical terms they become more explicit and more accessible for validation. If you can simulate a mathematical model that represents the phenomenon you are trying to understand, and compare data from it with data from experiments, then you can often know precisely where your theory diverges from reality and thus which assumptions were false. Or vice versa, if the simulation closely agrees with experiments, then that is a major indication that you have understood the mechanisms of the phenomenon correctly. This is the general motivation behind computational methods in neuroscience and any other scientific field.

The motivation behind this thesis specifically is to explore the capacity of the LFP to provide information about what a brain is doing. As we will discuss later, the LFP is a very compact representation of the activity of a brain region, and it is interesting to study how much information one can extract from this signal only.

Another interesting question to ask – after having trained the AI to predict external stimuli from the LFPs generated by simulations – is if we can then use the same AI to predict similar stimuli from LFPs generated by biological brains. If this would turn out to be the case, it would strongly imply that the simulation was a very good representation of the real thing, and hence that we had understood the system well. This is far beyond the scope of this thesis, but is an extension of this project that could possibly be done in a foreseeable future.

1.2 Goal and objectives

The goal of this project – in more technical terms – is to train AIs to predict the external stimuli to a simulated network of biological neurons, by looking at

the LFP that this network generates. We use a set of novel ways to simplify the calculations of the LFP signals, and the goal is partly to see how well we can reach our objective using these simplifications.

We begin by carefully verifying that that the model behaves as expected, using very simple external stimuli, and gradually make it more complex until we finally go on to stimulate it with a series of 10 images. We then test and compare how different types of algorithms are able to classify the data.

1.3 Additional notes

1.3.1 Notation and abbreviations

The tables below contains an overview of the notation and abbreviations we will use throughout this text.

Notation	
$\boldsymbol{a}$	Column vector
a_j^i	Element j of vector i
$\boldsymbol{A}$	Matrix operator

Abbreviations	
LFP	Local field potential
LGN	Lateral geniculate nucleus
LIF	Leaky integrate and fire (a neuron model)
AI	Artificial intelligence
ANN	Artificial neural network
DNN	Dense neural network
CNN	Convolutional neural network
LSTM	Long short-term memory
PCA	Principal component analysis
PSD	Power spectrum density
DOG	Difference-of-Gaussian
Presynaptic	The neuron whose axon the synapse is a terminal of
Postsynaptic	The neuron that the synapse in question connects to
L2 distance	Eucledian difference between two points or vectors

1.3.2 Note about the term "neural network"

An important thing to keep in mind is that throughout this text we will talk about two very different forms of networks of neurons that it is necessary to distinguish between. The whole text is divided along this distinction, in the way that the background and method sections are split into part I and II, one for each of the two network types.

1. When use the term **neuron network simulation**, we are talking about simulating the dynamics of a **biological** network of neurons with all (or at least some) of their physical features. This is in other words a simulation of a physical system which happens to consist of biological neurons.

2. When use the term **artificial neural networks**, we are referring to a certain set of algorithms used for optimization, i.e. **artificial intelligence**. These algorithms are inspired by how biological neuron networks work, but their goal is solely to solve a problem and in no way to capture the physics of real neurons.

The first of these algorithms is used to **generate** LFP signals, and the second one is used to **classify** them.

Chapter 2

Background I - Biological neural networks

In this chapter we go briefly through the physiology and physics of biological brains.

2.1 The neuron

Brains are made up of **neurons** – microscopic cells that communicate with each other using electrical signals. An average human brain consists of about 100 billion [2] of these units, and the number of connections is even more astronomical. A diagram of a neuron is shown in figure 2.1.

The neuron is covered by a **membrane** (figure 2.2 A) whose permeability varies for the different types of ions that are dissolved in the liquid medium inside the brain. Across the membrane we find **ionic pumps** – protein structures that force specific types of ions in or out of the cell [37]. This creates an ion concentration difference between the inside and the outside of the cell, which sets up an electric potential across the membrane. This is known as the **membrane potential** and is the basis for the electrical mechanics of the neuron [37].

If the membrane potential reaches a certain voltage level, it will trigger the membrane machinery to generate a sudden voltage increase, followed by a rapid decrease – i.e. a spike – which will propagate itself along the membrane. This is known as an **action potential** (figure 2.2 B) and is the signal that the neurons use to communicate with each other [37]. Neurons receive signals from other neurons on their **dendrites**, thin pipelines (covered by the same membrane) which can be thought of as their antennas. If a dendrite receives sufficiently many incomming spikes in a short period of time, it will itself generate a spike that will propagate inwards to the **soma** – the cell body of the neuron – where

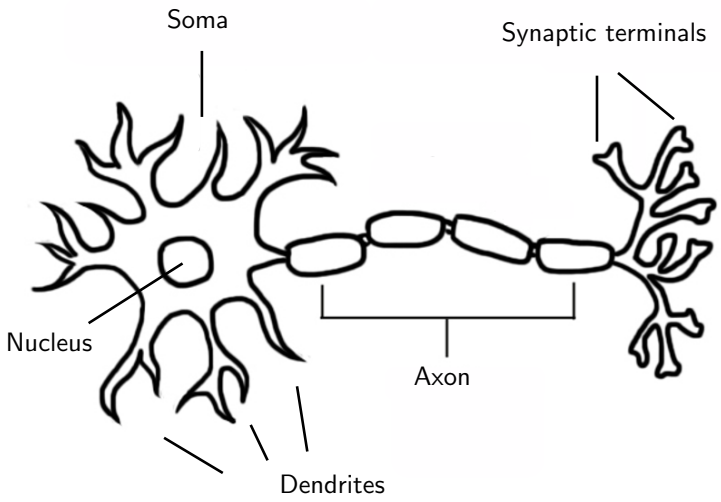


Figure 2.1: Illustration of a neuron. The cell body can range in size from 4 to 100 μm in diameter. Figure credit: www.brainfacts.org.

it will temporarily add to its membrane potential. If enough incomming signals push the voltage in the soma past the threshold, a spike will be generated and travel down the neuron's **axon** – a pipeline that ends in several terminals which connect it to the dendrites of other neurons. These terminals are called **synapses**, and when a spike reaches one it will activate the synapse and cause it to release **neurotransmitters** – chemicals that react with the membrane of other neurons by causing an in- or outflux of ions. As this affects the membrane voltage of other cells, this is how neurons are able to communicate.

Synapses that cause an increase in the postsynaptic potential are known as **excitatory** synapses, and the ones that lower it are known as **inhibitory** synapses. Individual neurons are commonly covered by synapses from thousands of other neurons, and if enough of these activate at around the same time they will cause the neuron to produce an action potential, following the mechanisms explained here.

There are many different types of neurons that differ in their mechanics and their function in the body. Some are triggered by external factors, for example tiny changes in air pressure, light of certain wavelengths or exposure to particular molecules. These are the ones responsible for our sensory perceptions [2] and can be found all over the body. Other neurons are connected to muscles and govern locomotion, reflexes and similar. But most neurons are simply one of many in large networks and work simply to react to others and get others to react to them. This is the topic of the next section.

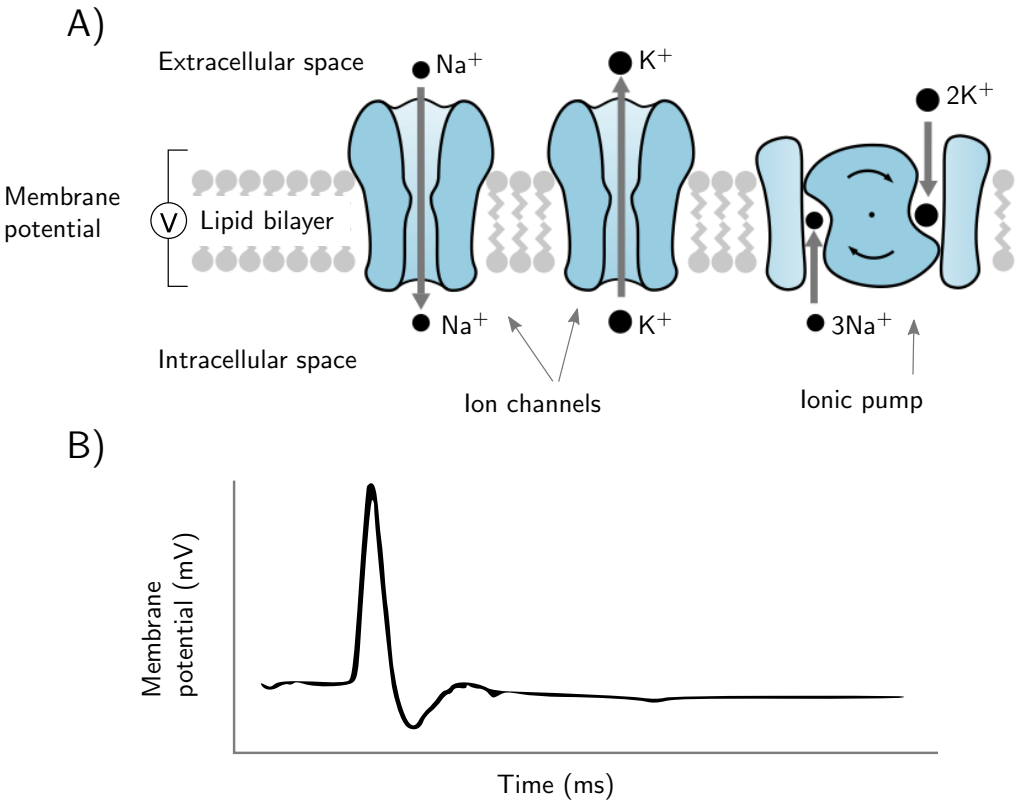


Figure 2.2: A) Illustration of the neuronal membrane. The lipid layer isolates the intracellular space from the extracellular space. Ion channels let specific types of ions diffuse through, and the pumps spend energy to pump ions in or out against the concentration gradient. **B)** The membrane potential of a neuron as it reaches the threshold value. This spike shape is what is known as the action potential. Figure adapted from [37].

2.2 Networks of neurons

When thousands and thousands of neurons are connected to each other, they form a system that can process information. The way this works is actually somewhat possible to understand:

If a specific subset of the neurons start firing for any reason, say for example that they recieve signals from touch sensors on the arm, they will trigger some other set of neurons to fire. This will then trigger certain others, which will trigger others again, and so on. It should seem reasonable that varying the connections and synapse strenghts will produce different such patterns of firing. In other words, by tuning the connections in a certain way, this domino of neural firings can be tuned such that it eventually reaches and triggers the specific set of neurons that control the muscles in the arm, such that it prompts them to

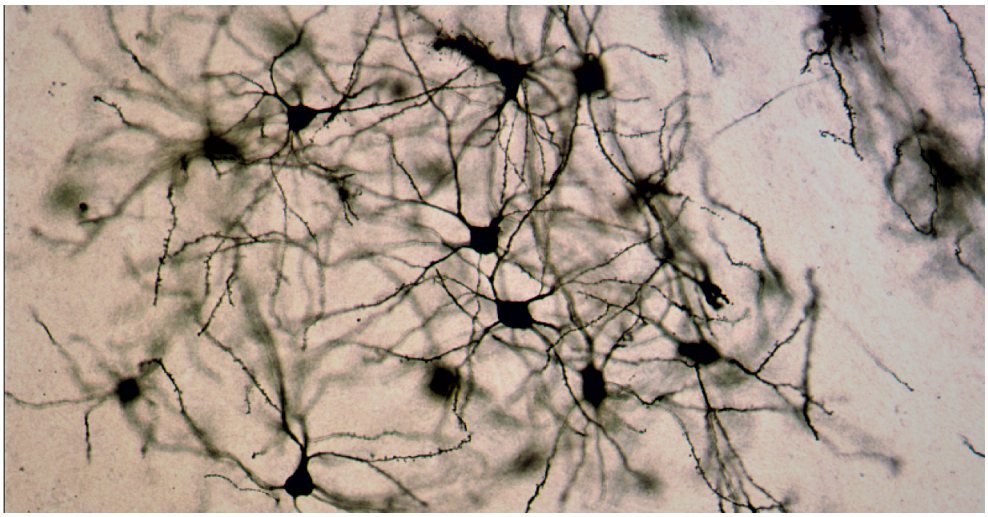


Figure 2.3: Network of biological neurons *in vivo*. These specific ones are of a type called *principal neurons* from the basolateral amygdala. The neurons are stained with the Golgi-method and then photographed through a microscope. Image credit: The Bergstrom Lab.

remove the arm from whatever it touched.

The more neurons there are, the more different ways there are to tune the network and thus the more complex the information processing can become. The brain with its billions of neurons works as a centralized system where different types of information about the external and internal environment are processed and combined in order to create extremely complex actions and reactions. The process needs to be highly complex, because the exact same input from the touch sensors on the arm might require very different responses in various contexts. For example, if the arm mentioned above was already intentionally reaching towards whatever it was that activated its touch sensors, then the desired output might be to pick up the object instead of just quickly removing the arm from it upon touch, which on the other hand might have been warranted if the simulus came as a surprise. Thus the information about that prior intention need to somehow be combined with the sense of touch in order to create a different outcome, and this is some of what the brain is doing.

The above mechanism is undoubtedly a very simplified explanation of how the brain works, but this simple, straightforward principle is what has been used to create artificial intelligence algorithms that are now superhuman in several fields. That does clearly indicate that this understanding is onto something, even though it might not be the only principle in working. There is for example evidence of other types of information processing going on [34].

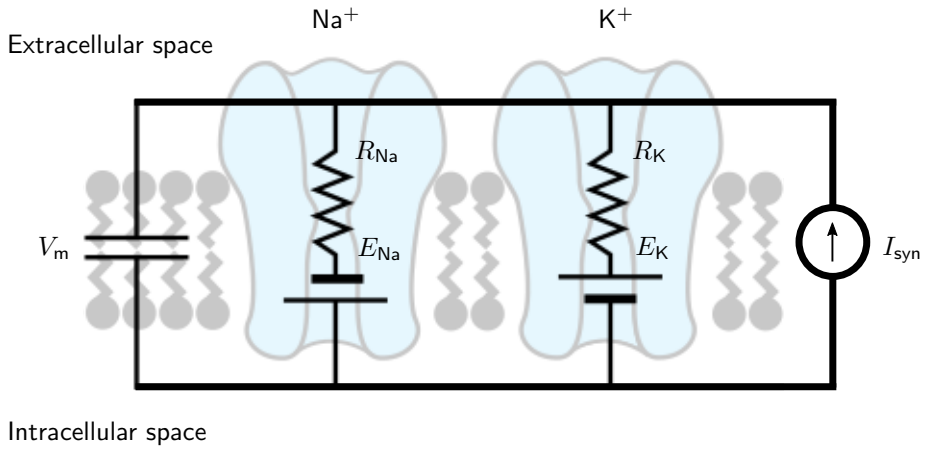


Figure 2.4: Illustration of how a patch of membrane is modeled as an electrical circuit. V is the membrane potential, E_{Na} and E_{Na} are the electromotive force for the Na and K ion channels and R_{Na} and R_K are the channel resistances. $I(t)$ is the synaptic input current to this patch of membrane. Figure adapted from [37].

2.3 Mathematical modeling of neurons

When making mathematical models we must decide upon some level of abstraction. There are many possible such levels when modeling neurons, and which one to pick will depend on several things – what the goal of the model is; how much computer power we have access to; how much time we have; how realistic the model needs to be, et cetera. We can think of it as a spectrum of abstraction where as we move upwards, we give up detail and realism for easier implementation and faster computation, and vice versa.

The simulation that we build in this project combines three different levels of abstraction. For simulating the early visual system we use a very simple rate-based model (see section 2.3.3 and 4.2). Then we use a slightly more detailed but still very simple model – a spiking neural network (section 2.3.1) – for simulating the activity of LGN and cortex. Lastly we use a more detailed one – a multicompartment model (section 2.3.2) – in order to calculate the LFP signals. More details on how and why we do it this way is found in chapter 4. The three models mentioned here are described in the next few subsections.

A general feature across most neuron models is to use the observation that the membrane potential is physically analogous to the voltage across a **capacitor**. One can then use appropriate equations from electromagnetism to describe its dynamics. The rate of change of the membrane potential $V(t)$ at time t can then

be expressed in terms of the ionic current $I(t)$ flowing in or out of the membrane, i.e.

$$C_{\mathrm{m}}\frac{dV(t)}{dt} = I(t), \tag{2.1}$$

where C_{m} is the membrane capacitance. This property is used for two of the neuron models used in this project.

2.3.1 Neuron model I – The LIF point neuron

The first model to mention is the leaky integrate-and-fire (LIF) point neuron model, which is the type we will use for the simulation of the cortical spiking activity. LIF models are one of the simplest ways to describe a neuron mathematically, and are often referred to as point neurons as their equations have no spatial dimension in them. The LIF neuron is modeled as a single RC circuit [37] through the equation

$$\tau_m\frac{dV(t)}{dt} = -V(t) + V_0 + R_{\mathrm{m}}I(t)(t), \tag{2.2}$$

where R_{m} is the membrane resistance, τ_{m} the membrane time constant, $I(t)$ the synaptic input current at time t and V_0 is the membrane resting potential – meaning its equilibrium potential when the synaptic input current is zero. Since we can choose reference points arbitrarily, we will for practical purposes set $V_0 = 0$ for the whole of this project.

Synapses are here included implicitly through the postsynaptic membranial currents that they generate when activated. The synapses of the LIF network will follow a delta function, in accordance with [1], expressed through

$$R_{\mathrm{m}}I(t)(t) = \tau_{\mathrm{m}}J\sum_k \delta(t - t_{\mathrm{s}}^k - \tau_{\mathrm{delay}}), \tag{2.3}$$

where t_{s}^k are the spike times of the particular neuron and τ_{delay} a variable that represents the time it takes for the action potential to propagate down the axon. In other words, if a neuron fires at time t, its synapses are activated at time $t + \tau_{\mathrm{delay}}$. The parameter J defines the **strength** of the synapse, which is the amount of voltage change that a single spike induces in the postsynaptic membrane potential. This means that $J > 0$ for excitatory synapses and $J < 0$ for inhibitory ones.

Implementing spikes

In order to include spiking activity into a network of LIF models, a threshold potential V_{thr} is chosen. Whenever $V(t)$ reaches this value for a given neuron, the cell will generate a spike and activate all its synapses. After spiking it will reset to a lower value V_{reset}, and will be insensitive to new synaptic input for a time period of τ_{ref}, known as the refractory period. This means that any input current $I(t)$ that is large enough to satisfy the criteria

$$R_{\text{m}} I(t) > V_{\text{thr}}, \tag{2.4}$$

will eventually make the neuron spike if it goes on for a sufficiently long time. If the input current also lasts after the refractory period is ended, the neuron will start building up potential again and generate another spike. This will keep going on as long as the input current lasts, and the larger $I(t)$ is, the quicker $V(t)$ will keep reaching V_{thr} and hence the higher the neuron's firing rate will be.

If we let a neuron receive **excitatory** input from a total of C synapses, in the form of spikes at constant rate ν, then we can state the synaptic input current as

$$R_{\text{m}} I(t) = CJ\nu\tau_{\text{m}}. \tag{2.5}$$

By substituting this into equation 2.4, we can solve for ν in order to get the **threshold rate**

$$\boxed{\nu_{\text{thr}} = V_{\text{thr}}/(JC\tau_{\text{m}})}, \tag{2.6}$$

as defined in [1].

A neuron that receives incomming spikes from at a higher rate than ν_{thr} will itself produce spikes. This parameter is therefore useful as a reference point for describing the amount of external input that the network receives, which we come back to in chapter 4.

2.3.2 Neuron model II – The multicompartment model

The second neuron model that we use in this project is the passive multicompartment model [37], which is a model that includes much of the physics that we need in order to calculate the LFP. The idea behind it is to model the neuron as a set of cylindric compartments where each cylinder has assigned to it a differential equation that describes the membrane potential of that specific part. As illustrated in figure 2.5, each compartment is modeled as an individual **RC circuit**, and the membrane potential $V_j(t)$ of compartment j at time t is

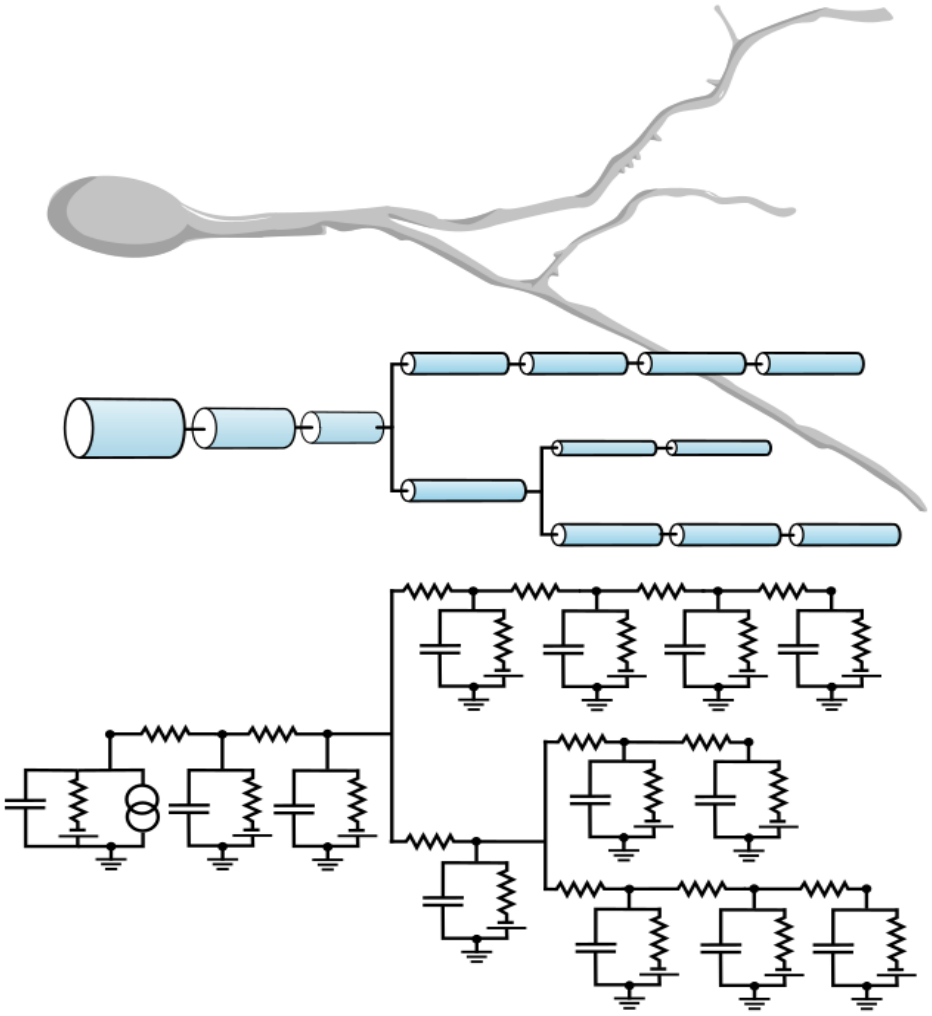


Figure 2.5: Illustration of the multicompartment model. Figure adapted from [37].

then found through the discretized **cable equation** [37]

$$C_{\mathrm{m}}\frac{dV_j}{dt} = g_j(E_j - V_j) + \frac{d_j}{4R_{j\mathrm{a}}}\Big(\frac{V_{j+1} - V_j}{\Delta s_j^2} + \frac{V_{j-1} - V_j}{\Delta s_j^2}\Big) + \frac{I_j}{\pi d_j \Delta s_j}. \tag{2.7}$$

The parameters d_j, Δs_j and $R_{j\mathrm{a}}$ are the diameter, length and axial resistance, respectively, of the given compartment. The middle term on the right side of the equation describes the axial currents flowing into the compartment from its neighbouring compartments, and I_j is the total synaptic input current to compartment j (see equation 2.10). E_j and g_j are the equivalent electromotive force and conductivity for all the ion channels and pumps in compartment j

combined, found through Thévenin's theorem [37] as

$$E_{\mathrm{j}} = \frac{\sum_x g_{xj} E_{xj}}{\sum_x g_{xj}} \tag{2.8}$$

and

$$g_j = \sum_x g_{xj} \tag{2.9}$$

E_{xj} and g_{xj} are the electromotive force and conductivity for the individual ion type x. As E_j and g_j are independent of V_j, this is where the model makes the assumption that the ion channels are **passive**, meaning that their permeability do not change in response to changes in the membrane potential – as opposed to **active** channels which do. The ionic pumps are modeled implicicly by assuming that they maintain the ion concentrations constantly through time [37]. Most channels in real neurons are active, so this is a major simplification [37], but it will suffice for our purpose.

Here too the **Synapses** are included implicitly through the postsynaptic membranial currents that they induce when activated. However, in contrast to the delta synapses that were used for the LIF neurons, the multicompartment model makes use of **alpha** synapses that follow the function

$$I_j(t) = Jte^{1-t/\tau_{\mathrm{syn}}}, \quad t > 0. \tag{2.10}$$

The equation describes the current induced in compartment j from a synapse activated at time t. The alpha synapse is used because it is more realistic, as the current transmition is distributed over time, more similar to how it is in biological neurons. The parameter τ_{syn} is the synaptic time constant that regulates the speed at which the synapse inputs the current. In the limit $\tau_{\mathrm{syn}} \to \infty$ this approaches the delta function. J, I and V represents the same thing as for point neuron model, however we need to scale the J of the alpha synapse so that it inputs the same amount of current as the delta synapse. This is detailed out in section 4.4.2.

The passive multicompartment model will not produce spikes (unless we implement it explicitly as for the point neurons), but this is fine for our purpose, as we will only use it to generate the LFP from already produced spiking patterns. More on why and how we do this is found in chapter 4.

2.3.3 Modeling the early visual system

Sensory stimulus of different kinds are processed in their own separate systems in the brain [2]. In this thesis we look at the system that processes visual

information. As implied in figure 1.1, we can for our purpose think of it as consisting of three stages, namely the eye, the LGN and the visual cortex.

LGN, which is an abbreviation for *lateral geniculate nucleus*, is a population of neurons that function as a relay station for information sent from the eye to the visual cortex. We will here outline a mathematical account of what happens between the first and second stage in this system. The model is very simplified in two significant ways: It is stripped of all the electrical dynamics, and it is **rate-based**, which means that it treats the firing rate of the neurons as the unit of information, as opposed to the individual spikes. It works as follows:

In the back of the eye – the retina – we find two types of photoreceptive neurons known as **rods** and **cones** [2]. When these are hit by rays of light of certain wavelengths, they react by firing action potentials. These signals are then sent to another set of retinal neurons known as the **retinal ganglion cells**. The ganglion neurons react to patterns of activity in the photoreceptive layer in a very specific way that has been found empirically to follow a **difference-of-Gaussian** (DOG) impulse-response function [31, 9]. What this means is that if we think of the incomming pattern of light as a 2-dimensional plane, which we for simplicity will assume is unicolored and stationary in time, then we can define a function $S(\boldsymbol{r})$ that describes the light intensity at point $\boldsymbol{r}$ in this plane, see figure 2.6. We define high values of $S(\boldsymbol{r})$ to correspond to points with high light intensity and low values to points with low light intensity. The response λ, i.e. the firing rate, of a specific ganglion cell to this pattern of light then follows the equation

$$\lambda = \iint_{\boldsymbol{r}'} W(\boldsymbol{r} - \boldsymbol{r}')S(\boldsymbol{r}')\ d^2\boldsymbol{r}', \tag{2.11}$$

where the impulse-response function $W(\boldsymbol{r})$ is, as its name implies, the difference of two 2-dimensional Gaussian functions

$$W(\boldsymbol{r}) = \frac{A}{\pi a^2} e^{-\boldsymbol{r}^2/a^2} - \frac{B}{\pi b^2} e^{-\boldsymbol{r}^2/b^2}. \tag{2.12}$$

The DOG can be visualized as the two concentric circles in figure 2.6 A with radius r_1 and r_2, centered at point $\boldsymbol{r}$ in the plane. Figure 2.6 B is a 1-dimensional version to show how the radii, which are functions of the parameters A, B, a and b, are defined to be where the values of $W(\boldsymbol{r})$ are practically zero. The DOG is how evolution has "designed" us to detect **edges** in the visual field, i.e. contours, and the following is an explanation of how that works.

First note that a contour is simply a sharp difference in contrast between two contigous regions. What the DOG does is to output high values of λ if the contrast in light intensity between the inner and the outer circle is great. If $A > B$, then λ is maximized if the inner circle is light and the outer circle is dark. And vice versa if $B > A$.

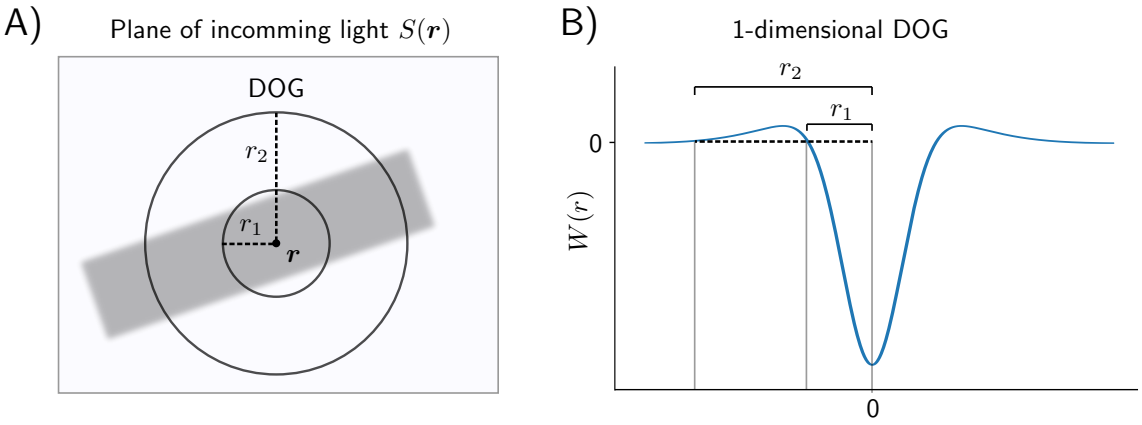


Figure 2.6: **A)** The difference-of-Gaussian (DOG) impulse-response function $W(\boldsymbol{r})$ shown as the two concentric circles. The darker parts represents points with low light intensity. **B)** A 1-dimensional DOG showing how the inner and outer circles corresponds to negative and positive values of W.

These two cases define, respectively, what are referred to as OFF-centre and ON-centre type cells in the literature [9]. The cell with the DOG in figure 2.6 would thus be an OFF-centre type.

To see how the DOG function achieves this behaviour, first note that all points outside of the biggest circle have zero contribution to the value of λ, as the value of $W(\boldsymbol{r})$ is zero for all these points. Looking next at the inner circle, we see that almost all points are dark, meaning that the values of $S(\boldsymbol{r})$ here are low. This is precicely what maximizes the integral in equation 2.11 over this region, since all the values of $W(\boldsymbol{r})$ here are negative. The opposite is true for the outer circle, as the values of $W(\boldsymbol{r})$ here are positive, so what maximizes the integral over bigger circle is that the points in S here have a high value, i.e. have a high light intensity. The example in 2.6A should in other words output a relatively high value of λ.

In this description of the early visual system we have only considered the spatial dimension. In reality the response of the synapses are not stationary in time, so we need to consider the temporal dimension when calculating λ as well. We will come back to this in section 4.2 when we talk about how we implement this model in the simulation.

To summarize this subsection: what the retinal ganglion cells are doing is to work as a high pass filter that detects contours in the visual field. This information is then sent to the LGN population, which forwards it to the visual cortex. Further filtering is happening in the LGN, but we will refer to [28, 8] for more on this.

2.4 The local field potential

When we want to analyse the activity of large networks with neurons that number in the hundreds of thousands and more, we cannot simply measure the activity of every single neuron. This motivates a more compact measure of brain activity, and luckily the electrical nature of neurons provide electromagnetic fields which can be used for this.

There are multiple ways to do this, a well known approach is the EEG, but for this project we will use what is known as the **LFP** – the Local Field Potential. This is defined as the low frequency part ($\lesssim$ 500 Hz [16]) of the electric field measured in the near vicinity of the neurons [7]. It is in other words what you will measure if you stick a tiny electrode into the brain, connect it to a voltmeter and apply a low pass filter to the signal you measure.

With volume conducting theory, we find that the electrical potential $\phi_{ij}(\boldsymbol{r}, t)$ from an individual transmembranial current $I_{ij}(t)$ of compartment j at position $\boldsymbol{r}'$ in neuron i, measured at time t by an electrode at point $\boldsymbol{r}$, is given as [25,20,16]

$$\phi_{ij}(\boldsymbol{r}, t) = \frac{I_{ij}(t)}{4\pi\sigma_e|\boldsymbol{r} - \boldsymbol{r}'|} \tag{2.13}$$

where σ_e is the conductivity of the extracellular medium. This is illutrated in figure 2.7. The full LFP, denoted as $\Phi(\boldsymbol{r}, t)$, is then found as the sum of ϕ_{ij} over all individual neurons and their compartments.

We assume that the current sources are uniformly distributed in space along each cylindrical compartment, so the total electric potential from comparment j of neuron i is found by integrating 2.13 along the cylindrical axis. The soma is modeled as a sphere, and its contribution is therefore proportional to the contribution from a point source at its center of mass. To find the extracellular potential for the full network, measured at point $\boldsymbol{r}$, we simply sum up the contributions from all axonal and dendritcal compartments of all neurons, plus the contribution for each soma [16], i.e.

$$\begin{aligned}\Phi(\boldsymbol{r}, t) &= \sum_{i=1}^{N} \frac{1}{4\pi\sigma_e}\left[\frac{I_{i,\text{soma}}(t)}{|\boldsymbol{r} - \boldsymbol{r}_{i,\text{soma}}|} + \sum_{j=2}^{n_\text{comp}} \int d\boldsymbol{r}'_{ij} \frac{I_{ij}(t)}{|\boldsymbol{r} - \boldsymbol{r}'_{ij}|}\right] \\ &= \sum_{i=1}^{N} \frac{1}{4\pi\sigma_e}\left[\frac{I_{i,\text{soma}}(t)}{|\boldsymbol{r} - \boldsymbol{r}_{i,\text{soma}}|} + \sum_{j=2}^{n_\text{comp}} \frac{I_{ij}(t)}{\Delta s_{ij}} \ln\left|\frac{\sqrt{h_{ij}^2 + r_{\perp ij}^2} - h_{ij}}{\sqrt{l_{ij}^2 + r_{\perp ij}^2} - l_{ij}}\right|\right].\end{aligned} \tag{2.14}$$

N is the total number of neurons and n_comp is the number of compartments in each cell. The soma counts as compartment 1, which is why j starts the iteration

at 2. Here $r_{\perp ij}$ is the norm of the component of $(\boldsymbol{r}-\boldsymbol{r}'_{ij})$ that is orthogonal to the compartment's cylindrical axis, and h_{ij} and l_{ij} are the norms of the component that is parallel to the cylindrical axis at the very beginning and very end of the integral, meaning $l_{ij} = \Delta s_{ij} + h_{ij}$.

This equation is based on the following assumptions [7]: First, that the electric field can be described by the quasistatic approximations of Maxwell's equations, which is appropriate for the frequencies seen in neural activity, and secondly that the extracellular conductivity σ_e is ohmic, frequency independent, homogeneous and isotropic (i.e. equal in all directions).

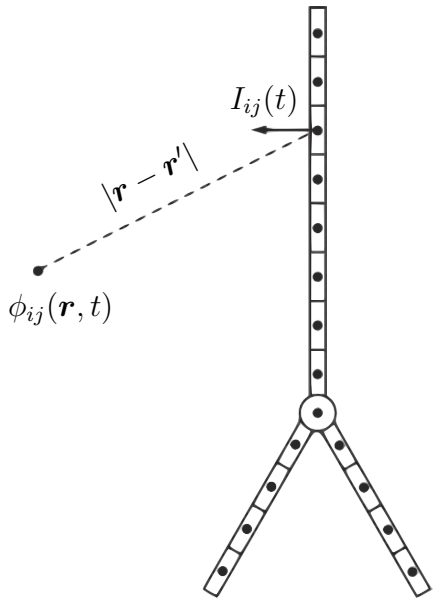


Figure 2.7: Illustration of how we calculate the LFP from the individual electric potentials from the transmembranial currents. Figure adapted from [7].

Chapter 3

Background II - Artificial neural networks

In this chapter we go through the theory behind a category of AI algorithms known as Artificial Neural Networks (ANNs) that we use for the classification part of this project. We outline three commonly used ones - the Dense Neural Network (DNN), the Convolutional Neural Network (CNN) and the Long Short-Term Memory (LSTM).

3.1 On artificial neural networks in general

Artificial neural networks (ANNs), also known as deep learning, is one of the most common type of artificial intelligence (AI) [27]. In short, it is simply a type of function that transforms an input vector $\boldsymbol{x}^i$ to an output vector $\hat{\boldsymbol{y}}^i$, i.e.

$$\mathcal{F}(\boldsymbol{x}^i) = \hat{\boldsymbol{y}}^i, \tag{3.1}$$

where the index i is just to separate individual datapoints.

$\mathcal{F}$ can also be defined to create outputs for a whole dataset at one time, written as

$$\mathcal{F}(\boldsymbol{X}) = \hat{\boldsymbol{Y}}, \tag{3.2}$$

where $\boldsymbol{X} = [\boldsymbol{x}^1 \dots \boldsymbol{x}^N]^T$ and $\hat{\boldsymbol{Y}} = [\hat{\boldsymbol{y}}^1 \dots \hat{\boldsymbol{y}}^N]^T$.

AI can be used for many purposes, but what we will use it for throughout this paper is **classification**. This means that we take an input $\boldsymbol{x}^i$ and assort it to a class index c^i. More specifically, we input an LFP signal to $\mathcal{F}$, and it outputs a set of predictions that represents how much it thinks the signal results from each of the 10 images – from here on referred to as **classes**. In other words, $\hat{y}^i_j$

represents $\mathcal{F}$'s confidence that $\boldsymbol{x}^i$ belongs to class j. The index of the highest element in $\hat{\boldsymbol{y}}^i$ is the AI's ultimate prediction, i.e.

$$\begin{aligned} \hat{c}^i &= \underset{j}{\operatorname{argmax}}\, \hat{\boldsymbol{y}}^i \\ &= \underset{j}{\operatorname{argmax}}\, \mathcal{F}(\boldsymbol{x}^i). \end{aligned} \tag{3.3}$$

There are many types of ANNs, but the common theme of them all is that they are in some way inspired by how the brain works, hence the name. The analogy between $\mathcal{F}$ and a brain should become more clear in the next section.

3.2 Dense neural networks

The Dense Neural Network (DNN) [27], also known as a Fully Connected Neural Network, is one of the simplest types of ANN. Moreover, it is a good basis from which to understand the more complex types of artificial neural nets. In its most consise mathematical formulation, a DNN can be written as

$$\mathcal{F}(\boldsymbol{x}^i) = g_L(\boldsymbol{W}^L\ g_{L-1}(\boldsymbol{W}^{L-1} \dots g_1(\boldsymbol{W}^1 \boldsymbol{x}^i) \dots)). \tag{3.4}$$

Each $\boldsymbol{W}^k$ is a matrix, and each g_k is a function $g : \mathbb{R}^l \to \mathbb{R}^l$, known as an **activation function**, that can take many forms but ultimately has the purpose of adding non-linearities to the transformation.

The two types of activation functions we will mainly see throughout this text are the **ReLu** function

$$g_{\text{ReLu}}(\boldsymbol{z}) = \begin{bmatrix} \max(0, z_1) \\ \max(0, z_2) \\ \vdots \\ \max(0, z_l) \end{bmatrix} \tag{3.5}$$

and the **Softmax** function

$$g_{\text{Softmax}}(\boldsymbol{z}) = \frac{1}{\sum_j^l \exp(z_j)} \begin{bmatrix} \exp(z_1) \\ \exp(z_2) \\ \vdots \\ \exp(z_l) \end{bmatrix}. \tag{3.6}$$

The analogy to the brain comes from thinking about $\mathcal{F}$ as successive layers of

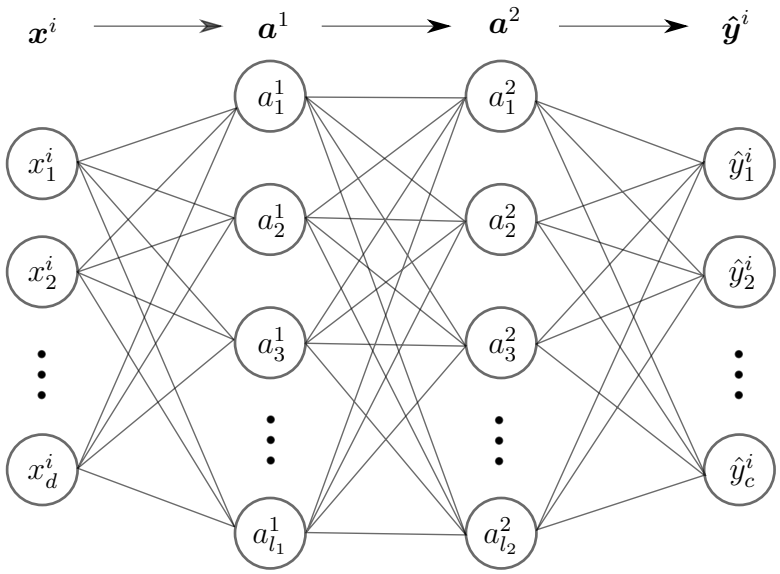


Figure 3.1: Illustration of a 4 layered DNN with input dimension d. The number of nodes are l_1, l_2 and c in layer 1, layer 2 and output layer, respectively. For simplicity, the superscript i is dropped from the hidden layers.

neurons which can be in active or inactive mode. The first layer would then be

$$\boldsymbol{a}^1 = g_1(\boldsymbol{W}^1\boldsymbol{x}^i), \tag{3.7}$$

the second layer would be

$$\boldsymbol{a}^2 = g_2(\boldsymbol{W}^2\boldsymbol{a}^1), \tag{3.8}$$

and so on until the last layer

$$\boldsymbol{a}^L = g_L(\boldsymbol{W}^L\boldsymbol{a}^{L-1}) = \hat{\boldsymbol{y}}^i \tag{3.9}$$

Each element in a vector $\boldsymbol{a}^k$ represents a neuron in the k'th layer, and the value of the element can be thought of as that neuron's firing rate. The value 0 would of course then be analogous to an inactive neuron. As the input vector counts as the zeroth layer, this makes up a DNN of $L+1$ layers.

From the dot products in equations 3.7 through 3.9, we can see that each element in $\boldsymbol{a}^k$ affects each element in $\boldsymbol{a}^{k+1}$. This captures the idea that all the neurons in a layer is connected to all the neurons in the next layer, as illustrated in figure 3.1. The $\boldsymbol{W}^k$ factors go under the name of **weight matrices**, and their elements represent the strengths of the individual connections between layer $k-1$ and layer k. Each matrix element thus corresponds to a single line in the figure above. Negative values in $\boldsymbol{W}^k$ are then analogous to inhibitory synapses.

A current popular convention is that the last layer uses g_{Softmax} as activation function while all the other ones uses g_{ReLu}, but other types can also be used.

3.3 Training a neural network

In order to get the function $\mathcal{F}$ to behave the way we want – meaning to correctly classify our datapoints – we need to tune its parameters right. A very common approach is to define a **loss function** $\mathcal{L}(\boldsymbol{x}^i, \boldsymbol{y}^i|\boldsymbol{\theta})$ that measures how well $\mathcal{F}$ is behaving, and then optimize that function. More specifically, $\mathcal{L}$ returns a scalar where a high value means $\mathcal{F}$ is tuned badly, while a low number means it is tuned well. So to optimize $\mathcal{F}$'s behaviour, we **minimize** $\mathcal{L}$. The term $\boldsymbol{\theta}$ represents all the tuneable parameters, meaning all the elements in the weight matrices $\{\boldsymbol{W}^k\}$.

We can minimize $\mathcal{L}$ by taking its gradient with respect to $\boldsymbol{\theta}$, and then move in the opposite direction in parameter space. (In other words we maximize $-\mathcal{L}$.) This is famously known as **backpropagation** [27], formulated as

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \eta \nabla_{\boldsymbol{\theta}_t} \mathcal{L} \tag{3.10}$$

where η is a parameter that can be tuned to regulate the size of each step in parameter space and t denotes the current cycle of training. We can equivalently state this in terms of each individual weight matrix

$$\boldsymbol{W}^k_{t+1} = \boldsymbol{W}^k_t - \eta \nabla_{\boldsymbol{W}^k_t} \mathcal{L}, \tag{3.11}$$

or we can write it in terms of each individual matrix element

$$w^k_{mn,t+1} = w^k_{mn,t} - \eta \frac{\partial \mathcal{L}}{\partial w^k_{mn,t}}. \tag{3.12}$$

The term $\boldsymbol{y}^i$ (not to be confused with $\hat{\boldsymbol{y}}^i$) in $\mathcal{L}$ is the **target output** for $\boldsymbol{x}^i$. It is vector that has a single 1-value placed at the index that represents the correct class for input $\boldsymbol{x}^i$, and the rest of the elements are zero. This is known as one-hot encoding. In other words, since we want $\mathcal{F}$ to make a correct classification, $\boldsymbol{y}^i$ is what we want $\hat{\boldsymbol{y}}^i$ to be. $\mathcal{L}$ should therefore somehow measure how close the two vectors are. There are several different forms of losses to choose from, but we will throughout this project stick to the commonly used **Cross Entropy Loss** function [27], defined as

$$\mathcal{L}(\boldsymbol{x}^i, \boldsymbol{y}^i|\boldsymbol{\theta}) = -\sum_{j=1}^{c} y^i_j \log \hat{y}^i_j, \tag{3.13}$$

where c is the number of possible classes. As we want the ANN to correctly classify every datapoint in our dataset $\boldsymbol{X} = [\boldsymbol{x}^1 \dots \boldsymbol{x}^N]^T$, we want to minimize

$$\begin{aligned} \mathcal{L}(\boldsymbol{X}, \boldsymbol{Y}|\boldsymbol{\theta}) &= -\frac{1}{N}\sum_{i=1}^{N} \mathcal{L}(\boldsymbol{x}^i, \boldsymbol{y}^i|\boldsymbol{\theta}) \\ &= -\frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{c} y_j^i \log \hat{y}_j^i, \end{aligned} \tag{3.14}$$

where N is the number of datapoints in the dataset and $\boldsymbol{Y} = [\boldsymbol{y}^1 \dots \boldsymbol{y}^N]^T$ are the one-hot encoded labels for all the data.

During training, $\mathcal{L}$ is calculated over a mini-batch of data for each forward pass. This is a small, stochastically sampled subset of the training set. The gradient descent is then performed using the update rule in equation 3.10 through 3.12, which for the mini-match case is just the average of all the individual gradients of all the individual data points in the mini batch. I.e.

$$\nabla_{\boldsymbol{\theta}}\ \mathcal{L}(\boldsymbol{X}, \boldsymbol{Y}|\boldsymbol{\theta}) = \frac{1}{N}\sum_{i=1}^{N} \nabla_{\boldsymbol{\theta}}\ \mathcal{L}(\boldsymbol{x}^i, \boldsymbol{y}^i|\boldsymbol{\theta}). \tag{3.15}$$

Understanding this should be sufficient to fully comprehend how a neural network can be trained to correctly classify data, and the full expression of the gradient can be derived using the chain rule and some appropriate algebra. However, most modern machine learning libraries (see section 5.2) will already have an automatic gradient computation built in, so the fine grained details of this are rarely needed in practice.

Intuition

To get an intuition for why optimizing equation 3.13 will make $\mathcal{F}$ better at classifying the data, we note that $\boldsymbol{y}^i$ is one-hot encoded and therefore the factor y_j^i has the binary values

$$y_j^i = \begin{cases} 1 \text{ for the correct class for } \boldsymbol{x}^i, \text{ meaning for } j = c^i \\ 0 \text{ otherwise.} \end{cases} \tag{3.16}$$

In other words, for a given i, the only term in the sum that will be non-zero, is for the single case $j = c^i$. Moreover, since $\hat{y}_j^i \in [0, 1]$, that means $\log \hat{y}_j^i$ will be a number between $-\infty$ and 0. Since there is an additional minus sign in equation 3.13, $\mathcal{L}$ will be a number between 0 and ∞. Thus, for a given i, tuning the weight parameters such that $\mathcal{L}$ gets smaller, means tuning $\mathcal{F}$ such that it gives a higher prediction for the correct class.

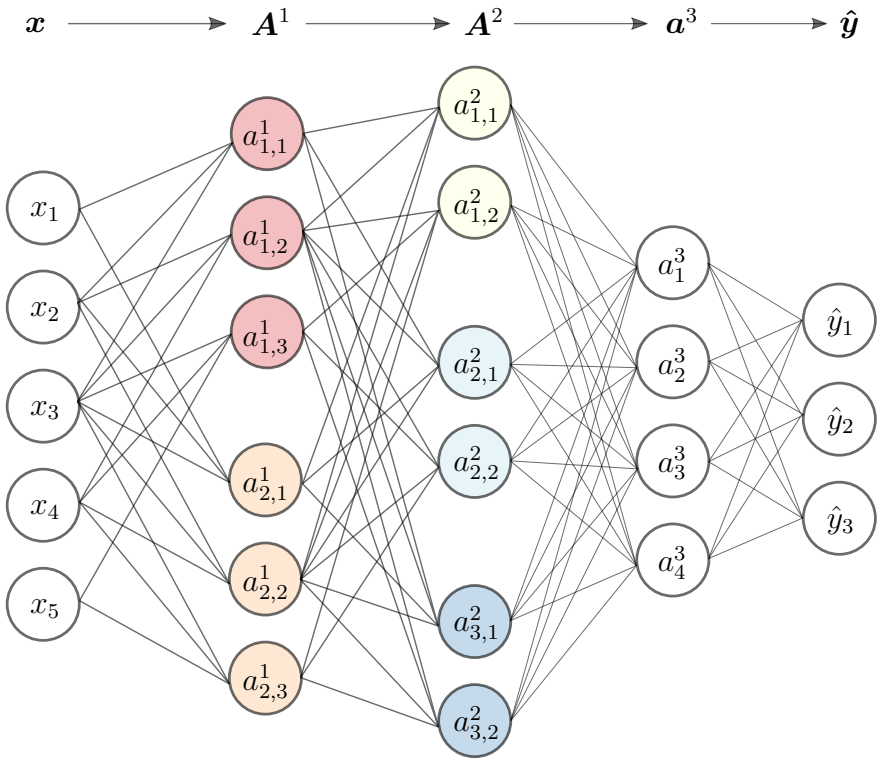


Figure 3.2: Illustration of a one dimensional CNN with two convolutional layers followed by two fully connected layers. The nodes in the convolutional layers are colored after which kernel (set of weights) they use. The first convolutional layers have two filters of size 3, the second has three kernels of size 2. It is easy to see how a CNN is a special case of DNN when illustrated this way. Note that the input layer has only one channel. With multiple channels in the input layer, the kernels in the first layer will connect to every input channel, just like the kernels in conv. layer 2 is connected to every channel in conv. layer 1.

3.4 Convolutional neural networks

A Convolutional Neural Network is a type of ANN that is inspired by the mechanisms of the visual system in biological brains (see section 2.3.3). Its main architectural purpose is to be invariant to the displacement of features in the input [27]. For example, if the inputs are images, we would want the network to respond virtually the same to an image with a cat in the top left corner as to an image with the same cat in the bottom right of the image. In our case the input data are one dimensional LFP timeseries, but similarly we want the network to have the ability to react to patterns independently of placement.

A CNN works much in the same way as a DNN, but contains something called **convolutional layers**. These are layers of nodes like in a DNN, but each node is here only connected to a small subset of the nodes in the previous layer. The

nodes in a convolutional layer are also grouped into **channels** such that every node in any one channel have the **same weights**. This ensures that any two nodes in the same channel that receives the same local input, responds exactly the same. The convolutional layers are usually followed by one or two dense layers. Figure 3.2 illustrates a CNN. Each column of nodes is one layer and the channels are differentiated by color.

A CNN can be viewed as a special case of DNN, in the sense that you can take a DNN, set most of its weights to zero and have groups of nodes share weights, and it will function as a CNN. In this sense we could have formulated it using a series of dot products as in equation 3.4. But as this would require extremely sparse weight matrices which would be highly computationaly inefficient, this is not how it is actually implemented in practice. It is therefore common to think of it as a series of **convolutions** with various **kernels** $\boldsymbol{W}$ (also called filters), which are matrices containing the sets of weights that the neurons in a channel share. Each channel in a convolutional layer therefore corresponds to its own such kernel.

Condensing all the equations down to a simple analytic expression, like equation 3.4, is a bit more tricky for a CNN. The mechanics of a convolutional layer k with N_k number of kernels can be written as

$$\boldsymbol{A}^k = \begin{bmatrix} \boldsymbol{W}^{k,1} * \boldsymbol{A}^{k-1} \\ \boldsymbol{W}^{k,2} * \boldsymbol{A}^{k-1} \\ \vdots \\ \boldsymbol{W}^{k,N_k} * \boldsymbol{A}^{k-1} \end{bmatrix}, \tag{3.17}$$

where $\boldsymbol{A}^k$ is a matrix containing the activations of layer k, and each row in $\boldsymbol{A}^k$ is a separate channel. The convolutional operator $*$ would here be defined as

$$\boldsymbol{W}^{k,q} * \boldsymbol{A}^{k-1} = \begin{bmatrix} \sum_{n=1}^{N_{k-1}} \sum_{m=1}^{F_k} w_{n,m}^{k,q} a_{n,m}^{k-1} \\ \sum_{n=1}^{N_{k-1}} \sum_{m=1}^{F_k} w_{n,m}^{k,q} a_{n,m+1S^k}^{k-1} \\ \sum_{n=1}^{N_{k-1}} \sum_{m=1}^{F_k} w_{n,m}^{k,q} a_{n,m+2S^k}^{k-1} \\ \vdots \\ \sum_{n=1}^{N_{k-1}} \sum_{m=1}^{F_k} w_{n,m}^{k,q} a_{n,m+D^k-F_k}^{k-1} \end{bmatrix}^T. \tag{3.18}$$

The kernels $\boldsymbol{W}^{k,q}$ are matrices of dimension $N_{k-1} \times F_k$. The term S^k is called the **stride** of layer k, and D^k is layer k's feature dimension size, meaning the number of points in the time dimension.

After performing this operation on all the convolutional layers in succession, the activation matrix $\boldsymbol{A}$ is flattened to a 1D vector and the rest of the layers work exactly as for a DNN. The CNN is similarly trained by optimizing the loss function in equation 3.14, altough the algrebra becomes a little more tricky. Again, we do not have to think about this, as `Keras` does it for us (section 5.2).

3.4.1 Receptive field of convolutional layers

The receptive field, also called the field of view, is an important concept when working with CNNs. In the literature, one separates between the **theoretical receptive field** (TRF) and the **effective receptive field** (ERF) [26].

The theoretical receptive field for a given node is the set of all input nodes that affect the value of that node to **any** degree. The size R^k of the theoretical receptive field in convolutional layer k can be calculated by the formula

$$R^k = R^{k-1} + (F^k - 1) \prod_{i=1}^{k-1} S^i, \tag{3.19}$$

where $R^0 = 1$, and F^k and S^k are the kernel size and stride of layer k, respectively. The stride is an integer that specifies the size of the jump that the kernel does each step in the convolution, and is often simply 1.

The ERF is defined as the set of input nodes that affect the given node to a **significant** degree, which will be a subset of the TRF. The fact that there is a difference between the TRF and ERF in a CNN stems from the fact that some of the input nodes will have many more pathways to reach any given node deeper down in the network than others. This means that some input nodes will factor into many more terms in the calculation of the given node's activation than other input nodes, thus having a much larger effect on the node's activation. This will usually follow a Gaussian distribution.

3.5 The Long short-term memory network

3.5.1 Recurrent neural networks

The Long short-term memory-algorithm (LSTM) is from a class of AI called Recurrent neural networks (RNNs), which we give a brief, general outline of here.

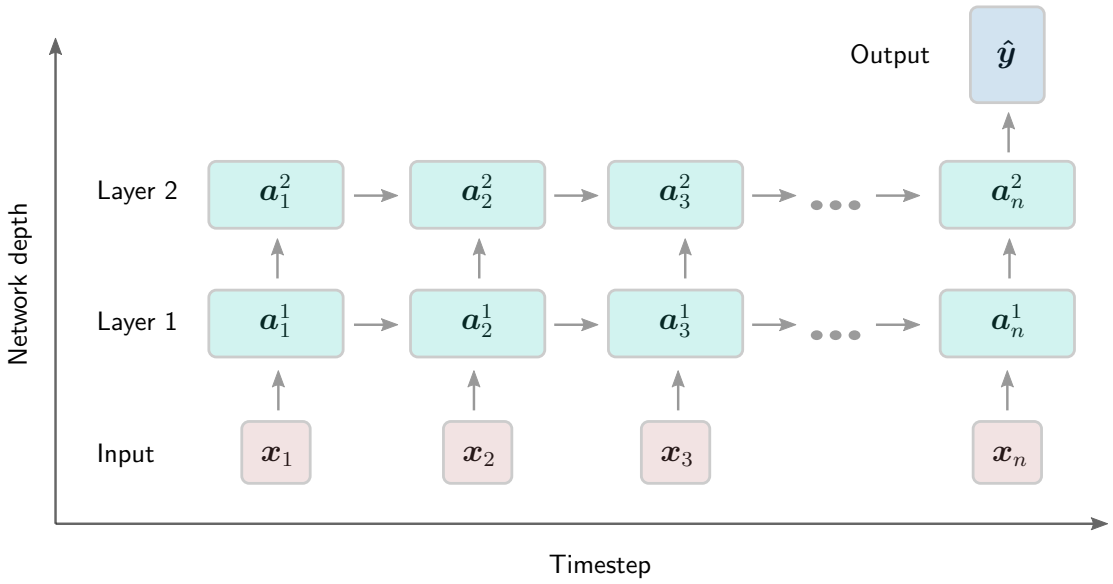


Figure 3.3: Illustration of the forward pass of an RNN, for example an LSTM, with two recurrent layers, used on input with n number of timesteps where one timestep is fed as input at a time.

RNNs are ANNs that contain layers of neurons that **affect themselves** over a time-like dimension – not only the next layer in succession. It is an extension of the simple feedforward principle behind networks like DNNs and CNNs, and tries to capture how populations of neurons in biological brains are interconnected. It is motivated by wanting models that can also take into account the **context** of an input, not simply the input itself. For example, if you want to make an AI that can read text, using individual words as input, then we want the network to take into account the other words in the sentence,not just the word that is the current input.

The feedforward algorithm of an RNN is divided into discrete steps, which we will for simplicity call timesteps. The activation of a recurrent layer $\boldsymbol{a}_t^l$ at timestep t, is affected by its activation $\boldsymbol{a}_{t-1}^l$ at the previous timestep. It is in addition affected by to layer $\boldsymbol{a}_t^{l-1}$, which is the layer one "spatial" step lower down in the network. This can be expressed by the general formula

$$\boldsymbol{a}_t^l = \mathcal{G}(\boldsymbol{a}_t^{l-1}, \boldsymbol{a}_{t-1}^l) \tag{3.20}$$

for some arbitrary function $\mathcal{G}$. If $l = 1$, meaning the first hidden layer, then the vector $\boldsymbol{a}_t^{l-1}$ is simply the input $\boldsymbol{x}_t$. As this notation indicates, the input can be different each timestep.

After calculating the output vector $\boldsymbol{y}$, the loss is calculated exactly as for the other ANNs. Training is, as before, performed by differentiating the loss function

and backpropagating.

The simplest form of recurrent neural net is the so-called **vanilla RNN**, [33] which uses a simple linear mapping of the two input vectors with some activation function g. A layer of a vanilla RNN is thus given as

$$\boldsymbol{a}_t^l = g(\boldsymbol{W}_1^l \boldsymbol{a}_t^{l-1} + \boldsymbol{W}_2^l \boldsymbol{a}_{t-1}^l). \tag{3.21}$$

3.5.2 The LSTM

The problem with the vanilla RNN is apparent when we calculate its gradient after many timesteps. We then have to perform the chain rule backwards in the time-like dimension, and this forces us to have to multiply the gradient many times with the same weight matrices. This has a tendency to cause the gradient to either vanish or blow up, which of course is very troublesome for training.

The Long Short-Term Memory (LSTM) network architecture [33] is designed to be a solution for this and is the current gold standard of RNNs. Instead of performing a single "linear-mapping-with-activation"-operation on the input, it employs a manifold of mappings in parallel and then applies additional operations on those again. This is done in order to combine them into the new state of the layer. These different transformation functions are often referred to as **gates**, and serve each their own specific function in deciding what information the layer should forget, what it should enhance, and so on.

The full mathematics of the LSTM is too comprehensive to fully include here, and figure 3.3 should provide a sufficient overall picture of how a general RNN works, so we will refer to [10,33] for the full details. The inputs $\boldsymbol{x}_t$ will in our case each be the set of LFP values at time t, one for each channel, and the output $\hat{\boldsymbol{y}}$ is of course, as before, the vector with the predictions for each of the 10 classes.

Chapter 4

Methods I - Simulating the LFP of a biological neural network

In this chapter we describe how we build a program that simulates a very simplified model of the visual system in the brain. The input to the model will be images and the output will be LFP signals.

4.1 Overview of the simulation

The full simulation consists of three separate parts, each using its own Python module. We describe this process very briefly here in this first section. The full details are found in the rest of this chapter.

Part 1. First we transform an input image to a **rate profile**. This represents the average firing rate of the neurons in the LGN population as a function of time, as they respond to the input. To do this we use a package called `pyLGN` [28].

Part 2. We then use the LGN rate profile as input to an inhomogeneous poisson generator. This creates **spike trains** with an average rate that corresponds to the rate profile. These spikes are then used as excitatory stimuli for a **point neuron network** that represents our very simplified cortex model. The point neuron network is built using the `NEST` module [15].

Part 3. Lastly the spiking activity from the point neuron network is used as input to a third module, which **estimates the LFP** based on the spiking pattern. This module is called `hybridLFPy` [16].

A schematic illustration of the simulation is shown in figure 4.1.

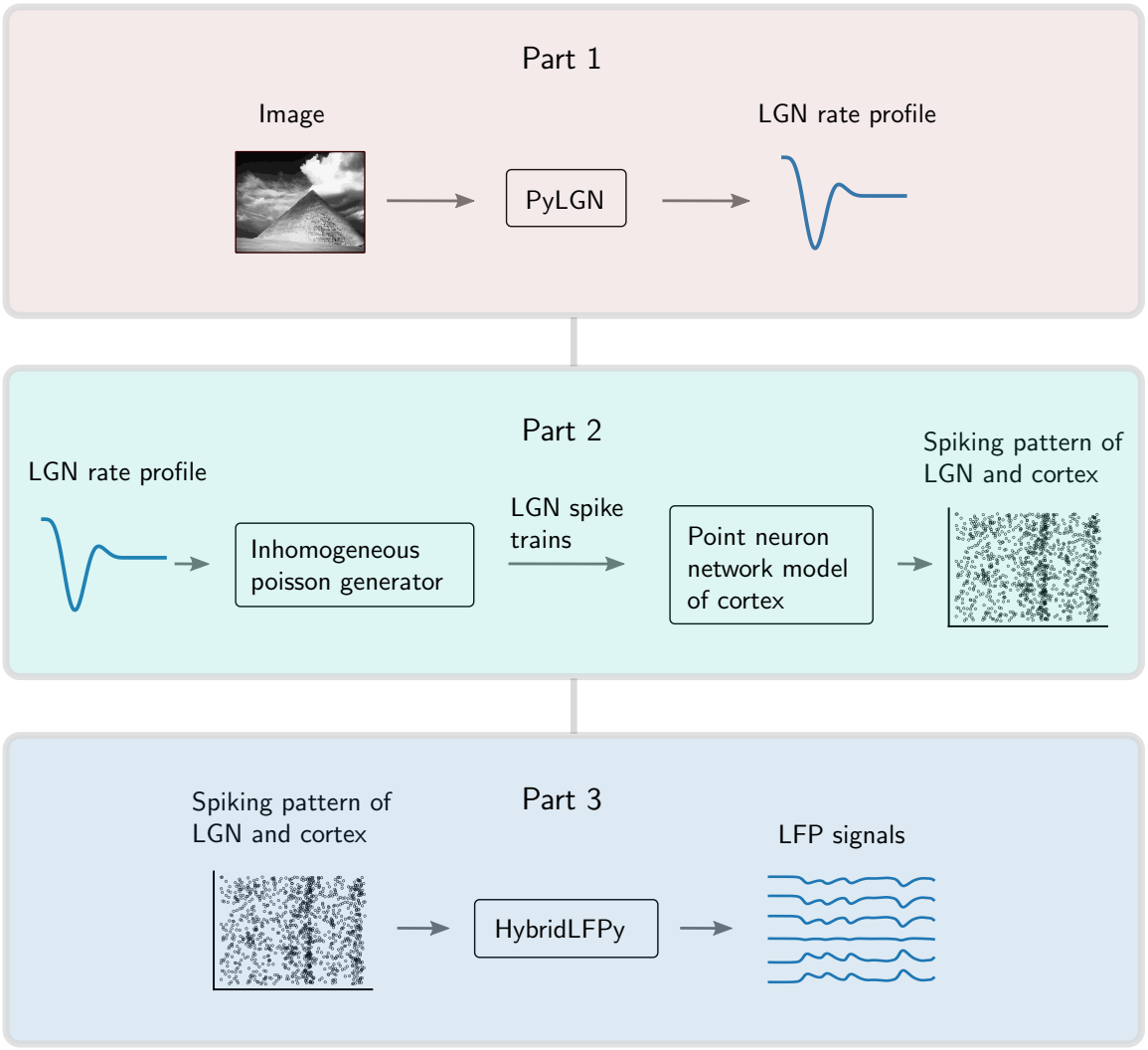


Figure 4.1: Schematic illustration of how the full simulation is built up to transform images to LFP signals.

The idea behind this approach is to combine the best of two worlds – the speed and simplicity of a point neuron network, and the physical properties of a multicomparmental model.

Point neuron networks alone do not include enough physics to allow for LFP computation. By that we mean that, since a LIF neurons with all its transmembranial currents is collapsed into a single point in space, the current sources and sinks that are the origins of the LFP [16, 7] cancel each other out. Consequently, calculating the LFP from a neuron network simulation would usually require us to run the full network simulation using multicomparment. We would need detailed morphologies and have to include models of active channels in order to get spiking activity. This is extremely computationally demanding,

and the way we do it here is designed to be a shortcut through that problem.

An important assumption behind this approximation is that the physics of the neurons has a small effect on the spiking activity. In other words, that the the spatial topology of the network, the shape of the action potentials, the LFP, et cetera, do not influence the spiking pattern very much. These are obviously not totally valid assumptions, but the degree to which they are is interesting to explore. There is for example evidence that the LFP can depolarize membrane potentials and in the extremes even cause spiking (e.g. in nerves damaged by multiple sclerosis) [20], but under normal circumstanses this is likely a very rare phenomenon.

4.2 Part 1 – Image-to-firing-rate with pyLGN

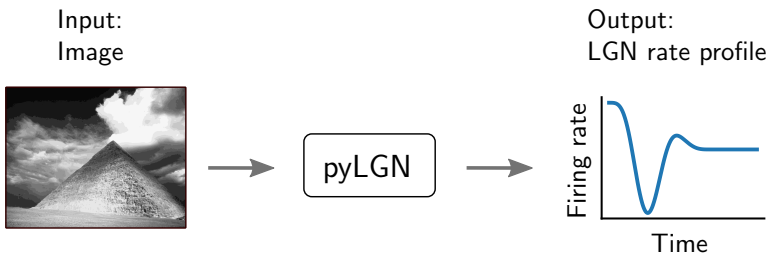


Figure 4.2: Illustration of part 1 of the simulation.

The Python module `pyLGN` is a deterministic, firing rate based model of the early stages of the visual system. It models an eye that receives optical stimuli on the retina, and sends this information to a part of the brain known as the LGN – the Lateral Geniculate Nucleus. LGN is a population of neurons that works as a relay station for visual information on the way to the visual cortex [28]. The module is developed primarily to study feedback effects, but for our purpose we will only use it as a feedforward tool to map an image to a rate profile. As said earlier, this will represent the average rate of neural firings in LGN as a function of time. For a description of the full module, see [28].

The feedforward part of `pyLGN` is on the empirical observation that retinal ganglion cells respond to visual stimuli in a very particular way. In more technical terms, they follow a difference-of-Gaussian impulse-response function [31, 9, 28], which from here on will be referred to as a **DOG**. The following is a description of how this works, and how it can be described mathematically.

1. We define a layer of neurons in a grid $R_{\mathrm{G}}(\boldsymbol{r}, t)$, where each point $\boldsymbol{r}$ represents a **retinal ganglion cell**, and the value at that point represents that cell's **firing rate** at time t. We define $\boldsymbol{r} = \mathbf{0}$ to be the center of the grid.

2. In a similar way, we can think of the input image as a grid of photoreceptive neurons on the retina, stimulated by light after a pattern that corresponds to the image. We denote this grid by $S(\boldsymbol{r}, t)$. The pixel at point $\boldsymbol{r}$ in $S(\boldsymbol{r}, t)$ then represents a single **photoreceptive cell**, and the scalar value at that point represents that neuron's firing rate at time t, as the neuron reacts to the photons hitting it. (We operate with grayscale images so the pixel only has a single value.)

3. We can now calculate responses of each neuron in the grid of ganglion cells R_{G} to the grid of photoreceptive cells S, by convoluting the image using the defined input-response function $W(\boldsymbol{r}, t)$. This is written as

$$R_{\mathrm{G}}(\boldsymbol{r}, t) = \int_{\tau} \iint_{\boldsymbol{r}'} W(\boldsymbol{r} - \boldsymbol{r}', \tau) S(\boldsymbol{r}', t - \tau) \; d^2\boldsymbol{r}' d\tau \tag{4.1}$$

or more compactly as

$$R_G(\boldsymbol{r}, t) = W(\boldsymbol{r}, t) * S(\boldsymbol{r}, t) \tag{4.2}$$

with $*$ as the convolution operator.

4. We assume that the impulse-response function W can be separated into a spatial and a time dependent part. The spatial part is modeled as a DOG, meaning literally the difference of two Gaussian functions, and its time dependence is factored in as a biphasic temporal function. In other words

$$W(\boldsymbol{r}, t) = F(\boldsymbol{r}) H(t) \tag{4.3}$$

where the spatial part is

$$F(\boldsymbol{r}) = \frac{A}{\pi a^2} e^{-\boldsymbol{r}^2/a^2} - \frac{B}{\pi b^2} e^{-\boldsymbol{r}^2/b^2} \tag{4.4}$$

and the time dependent part is

$$H(t) = \begin{cases} \sin(\pi t/\tau_{\mathrm{G}}), & 0 \leq t \leq \tau_{\mathrm{G}} \\ C \sin(\pi t/\tau_{\mathrm{G}}), & \tau_{\mathrm{G}} \leq t \leq 2\tau_{\mathrm{G}} \\ 0, & \text{otherwise.} \end{cases} \tag{4.5}$$

A, a, B, b are parameters describing the width and strength of the DOG, τ_{G} is the duration of each phase, with C as the weight for the second phase [28].

5. The last step is to compute the response of the LGN population in response to the activity of the ganglion population. Similarly as before, we define a

layer of cells $R_{\text{LGN}}(\boldsymbol{r}, t)$, where each point $\boldsymbol{r}$ represents a single LGN relay cell and the value at that point is the cell's firing rate at time t. We then need to define a **coupling kernel** $K(\boldsymbol{r}, t)$ that describes how the relay cells respond to the activity of the layer of ganglions, analogous to the impulse-response function $W(\boldsymbol{r}, t)$ from earlier.

Again we assume spatial and temporal independence. The spatial part of K is defined as a Gaussian, reflecting the fact that neurons close to each other are more likely to be connected, and the time dependent part is modeled as a (delayed) exponential decay. This is in accordance with previous modeling studies [28, 29, 6]. In other words,

$$K(\boldsymbol{r}, t) = wf(\boldsymbol{r})h(t) \tag{4.6}$$

where

$$f(\boldsymbol{r};\ d) = \frac{1}{\pi d^2} e^{-\boldsymbol{r}^2/d^2} \tag{4.7}$$

and

$$h(t;\ \Delta, \tau_{\text{LGN}}) = \frac{1}{\tau_{\text{LGN}}} e^{-(t-\Delta)/\tau_{\text{LGN}}} \theta(t - \Delta), \tag{4.8}$$

where w is the connection weight of the kernel, d is the width parameter, τ_{LGN} is the time constant and $\mathcal{H}$ the Heaviside function. The scalar Δ corresponds to a combined axonal and synaptic time delay [28].

R_{LGN} is now found as

$$\begin{aligned} R_{\text{LGN}}(\boldsymbol{r}, t) &= \int_\tau \iint_{\boldsymbol{r}'} K(\boldsymbol{r} - \boldsymbol{r}', \tau) R_{\text{G}}(\boldsymbol{r}', t - \tau)\ d^2\boldsymbol{r}'\ d\tau && (4.9)\\ &= K * R_{\text{G}} && (4.10)\\ &= K * (W * S). && (4.11) \end{aligned}$$

The way this is implemented in `pyLGN` is by using the fact that these operations can be done more easily in Fourier space, but there is no need to include that here. Again, we refer to [28] for more details on this.

Summary: We have here described how `pyLGN` maps an image to a set of firing rates that represents LGN cells that responds to visual input. An interesting thing to note is that this feedforward algorithm is very analogous to how a convolutional neural net (CNN) works from section 3.4.

For this project we will only use the rate profile of a single point in R_{LGN},

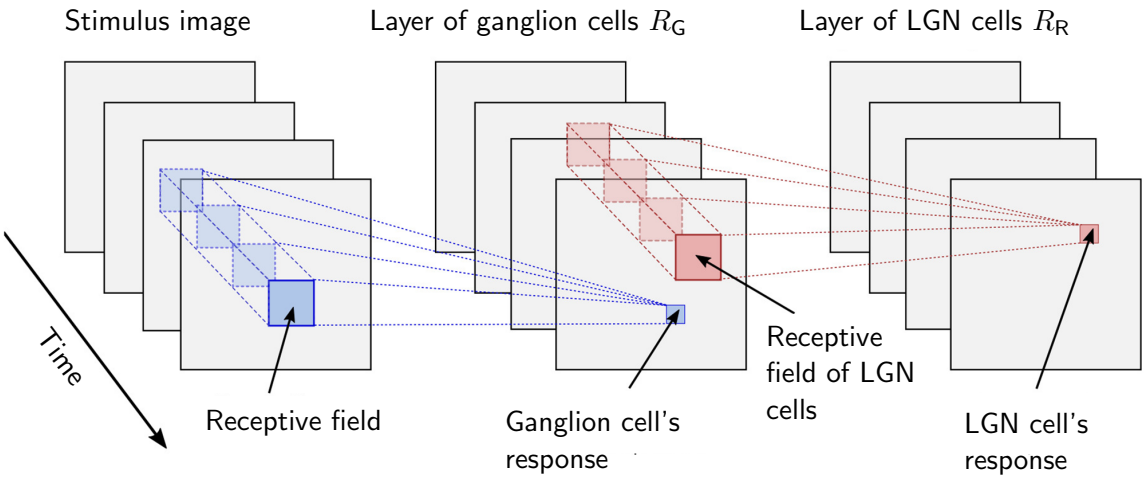


Figure 4.3: The feedforward algorithm of `pyLGN`. The model is very analogous to a 2 dimensional convolutional neural network. Figure adapted from [28] with courtesy of Milad H. Mobarhan.

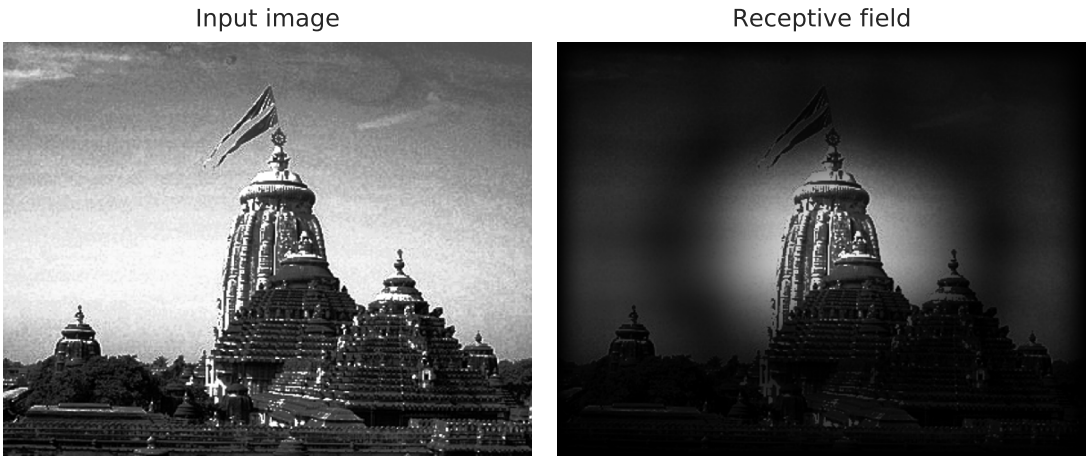


Figure 4.4: With the default parameters the receptive field of the center neuron in `pyLGN` that we use as rate profile for the LGN population is shown in the right image. Darker areas means less responsivity.

namely the one at the center. We can write this as

$$\boxed{\lambda_{\text{LGN}}(t) = R_{\text{LGN}}(\mathbf{0}, t)}. \tag{4.12}$$

$\lambda_{\text{LGN}}(t)$ is in other words the rate profile that we will use. This will function as the average firing rate for the whole LGN in the point neuron network simulation. The part of the input image that has an actuall effect on the center neuron is visualized in figure 4.4. We will referred to as the receptive field of the LGN population.

4.3 Part 2 – Point neuron network with NEST

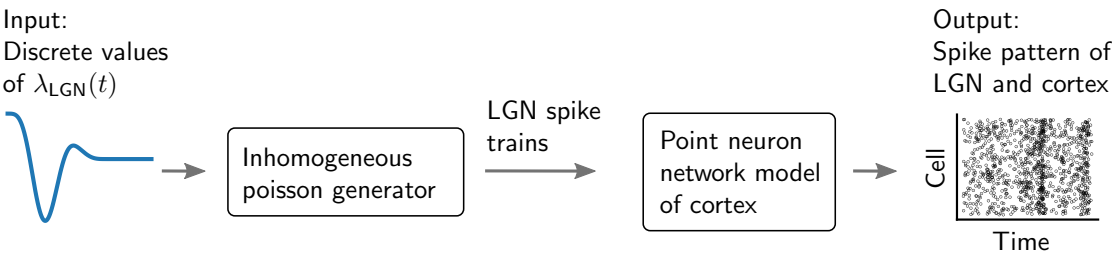


Figure 4.5: Illustration of part 2 of the simulation.

For the point neuron simulation, we use the Python module `NEST` [15]. It is an open-source tool designed for simulating large networks of point neurons in an easy and computationally cost effective way. It provides many different options of neuron models, synapse types and such. For our purpose we use a network consisting of LIF neurons, as described in section 2.3.1.

The network simulation consists of three populations. One representing LGN, and the two others representing cortex, as visualized in figure 1.1 and 4.6. The LGN population consists of N_{LGN} excitatory neurons, and cortex consists of N_{EX} excitatory and N_{IN} inhibitory neurons in their own respective populations. As in [1], we let the ratio of excitatory to inhibitory neurons conform to anatomical estimates from neocortex, so that 80% of the cortical neurons are excitatory, i.e. $N_{\text{EX}}/N_{\text{IN}} = 4$.

Each cortical neuron receives excitatory input from a number C_{LGN} of synapses from the LGN population. This comes in the form of Poissonian spike trains with an average rate that varies in time according to the rate profile $\lambda_{\text{LGN}}(t)$ from equation 4.12. The number of synapses from LGN is defined as

$$C_{\text{LGN}} = \epsilon N_{\text{LGN}}, \tag{4.13}$$

where $\epsilon = 0.1$ throughout this project. The cortical neurons in both populations

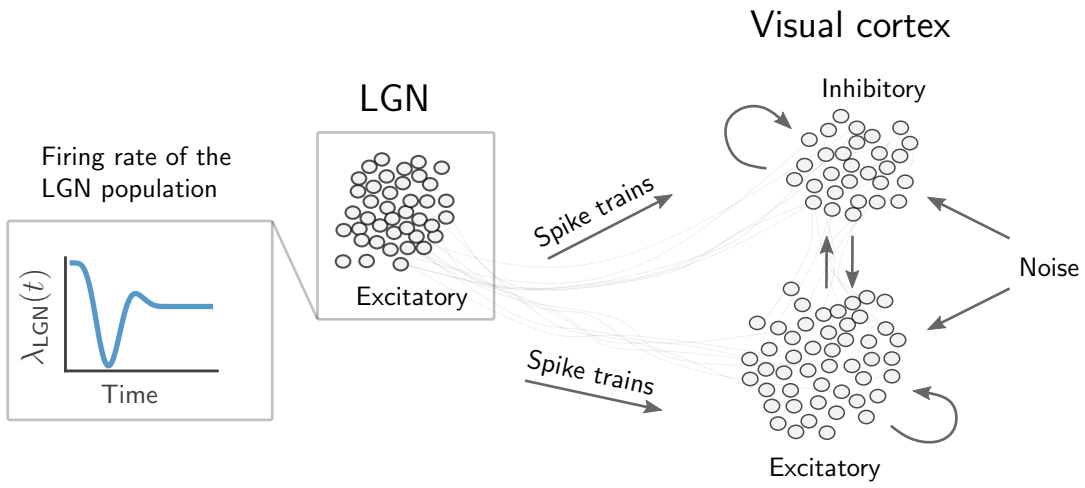


Figure 4.6: The point neuron network model.

also receieve spikes from other cortical neurons. We define C_{EX} and C_{IN} as the number of excitatory and inhibitory synapses, respectively, on each neuron in cortex. Similarly as before, we define these as

$$C_{\mathrm{EX}} = \epsilon N_{\mathrm{EX}} \tag{4.14}$$

and

$$C_{\mathrm{IN}} = \epsilon N_{\mathrm{IN}} \tag{4.15}$$

This way of defining connections is implemented with the `'fixed_indegree'`-rule in `NEST`'s `Connect()` function. Eq. 4.13, 4.14 and 4.15 are then used as the indegree values. In addition, every cortical neuron receive Poissonian spike trains from C_{BG} synapses at constant rate ν_{BG} , representing static background noise from the activity of the rest of the brain. Note, however, that in a point neuron network, receiving signals from C_{BG} synapses at a rate ν_{BG} is equivalent to receiving signals from a single synapse at a rate $\nu_{\mathrm{BG}} C_{\mathrm{BG}}$, as the locations of the synapses are all at the same point. We have therefore implemented the background noise in this way, as it is more computationally efficient.

4.3.1 Neuron and synapse models in NEST

The LIF neurons are implemented using `NEST`'s `iaf_psc_delta` model. Incomming action potentials change its postsynaptic membrane potential as a discrete jump – a delta function – and its subthreshold membrane dynamics follow equation 2.2.

The **synapse strength** is denoted as J_{EX} for all the excitatory neurons, and J_{IN} for the inhibitory ones. This parameter is defined as the amount that a single spike changes the postsynaptic membrane potential, and should thus be interprated to come in units of mV. As in [1] we define a parameter g to be the ratio of excitatory and inhibitory synapse strength, and define J_{IN} in terms of this. I.e.

$$J_{\text{IN}} = -gJ_{\text{EX}}, \tag{4.16}$$

where g is a value somewhere around the ratio $N_{\text{EX}}/N_{\text{IN}}$. This means that the total charge output from both populations are close to being on balance, despite the excitatory population being much larger.

As described in chapter 2, the neuronal firing mechanism is implemented in a LIF network through a **threshold value** V_{thr}. If the membrane potential of a neuron reaches this value, its synapses will be activated and its membrane potential will be reset to a value V_{reset}, after a **refractory period** τ_{ref} where all arriving spikes are simply discarded.

The **axonal delay** – the time that the action potentials use to travel down the axon – is modeled through a simple delay parameter τ_{delay}. If a neuron fires a signal at time t, its synapses will be activated at time $t + \tau_{\text{delay}}$.

4.3.2 From firing-rate to spikes

When biological neurons fire action potentials at a certain rate, the signals are not uniformly distributed in time but express some degree of stochasticity [3, 17]. This should be included in the model. To do this we will use an inhomogeneous Poisson generator to create stochastic spike trains that correspond to the $\lambda_{\text{LGN}}(t)$ rate profile. An explanation of the Poissonian process and the algorithm that we will use follows here.

The Poisson Process

A stochastic process that produces independent spikes at an average rate λ, can be viewed as there being a probability p every time interval $\Delta t = \frac{1}{n}$ that a spike is created, found as

$$p = \frac{\lambda}{n}, \tag{4.17}$$

where n is time resolution, i.e. the number of intervals per unit of time.

If the process starts at time $t_0 = 0$, then the probability that the first spike is created after a time $t = N\Delta t$ is equal to the probability that it *is not* created

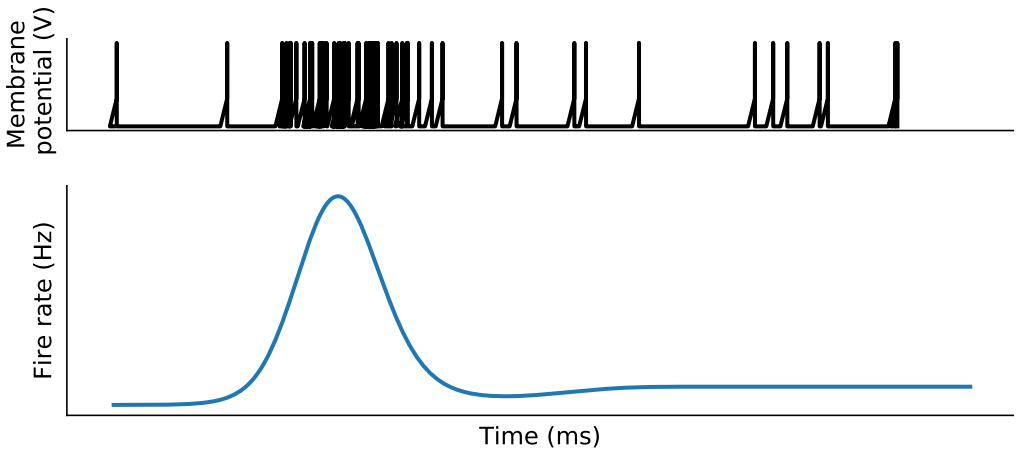


Figure 4.7: The upper chart shows an ihhomogeneous Poissonian spike train with a rate profile corresponding to the curve in the lower chart.

in the N first time intervals. The probability that a spike is not created in any given interval is of course $(1-p)$, so the probability that it does not occur after N intervals is therefore $(1-p)^N$, giving us the expression

$$P(\text{spike after } t) = (1-p)^N. \tag{4.18}$$

Substituting $N = \dfrac{t}{\Delta t} = nt$ and $p = \dfrac{\lambda}{n}$ gives us

$$P(\text{spike after } t) = \left(1 - \frac{\lambda}{n}\right)^{nt}, \tag{4.19}$$

and in the limit of perfect time resolution $n \to \infty$ we get

$$P(\text{spike after } t) = \lim_{n\to\infty} \left(1 - \frac{\lambda}{n}\right)^{nt} \tag{4.20}$$

which we see is the definition of the exponential function, i.e.

$$\boxed{P(\text{spike after } t) = e^{-\lambda t}.} \tag{4.21}$$

The probability that a spike occurs after a time t is in other words exponentially decreasing as t gets larger. To put it another way, equation 4.21 describes the probability distribution of time intervals in a Poisson process, and this is the basis of the spike generator algorithm.

The homogeneous spike generator

When the average rate λ is constant in time, we call it a **homogeneous**, or **stationary**, Poisson process. We begin with this case here as the algorithm for the inhomogeneous case is an extension of this. The whole idea is to solve equation 4.21 for t and use this to sample time intervals from the distribution using the principle of inverse transform sampling [5]. It works in the following way.

1. We begin by solving equation 4.21 for t as

$$t = -\frac{\ln U}{\lambda}, \tag{4.22}$$

   where we have rewritten $U = P(\text{spike after } t)$ for simplicity.

2. By drawing a random value U_1 from a uniform distribution between 0 and 1 and inserting this into equation 4.22, we are sampling a time interval Δ_1 from the exponential distribution in equation 4.21. In other words

$$U_1 \sim \text{Unif}\big(0, 1\big) \tag{4.23}$$

   and

$$\Delta_1 = -\frac{\ln U_1}{\lambda}, \tag{4.24}$$

   which is a positive number since $U_1 < 1$. We now have the interval between t_0 and the first spike, and can find the spike time as

$$t_1 = t_0 + \Delta_1 \tag{4.25}$$

3. The next spike time t_2 can then be found using the exact same method. We draw a random number U_2

$$U_2 \sim \text{Unif}\big(0, 1\big) \tag{4.26}$$

   to sample the second interval

$$\Delta_2 = -\frac{\ln U_2}{\lambda}. \tag{4.27}$$

   and get the next spike time

$$t_2 = t_1 + \Delta_2. \tag{4.28}$$

4. We repeat this process until we reach $t_{m+1} > T$, where T is the period we want the spike train to last. We now have a set of Poisson distributed spike times $\{t_1, t_2, .., t_m\}$.

The inhomogeneous spike generator

We will now explain the algorithm for generating Poissonian spike trains with a rate that varies in time – i.e. the **inhomogeneous** case. We assume that we know the full rate profile $\lambda(t)$ beforehand.

The idea is to first generate a stationary Poissonian spike train $\{t_1, t_2, .., t_m\}$, as explained in the previous subsection, at a rate $\lambda_{\max} = \max_t \lambda(t)$, and then stochastically eliminate subsets of this set at periods when the firing rate should be lower than $\lambda_{\max}$. This elimination process is called thinning. In practice we do not actually have to generate the full stationary spike train before we do the elimination part, but we can instead do both simultaneously. The algorithm goes as follows.

1. Starting at $i = 1$ we first calculate
$$\lambda_{\max} = \max_t \lambda(t). \tag{4.29}$$
2. We then as before sample a number U_i
$$U_i \sim \text{Unif}\big(0, 1\big). \tag{4.30}$$
3. We use this to sample a time interval from the interval distribution at rate $\lambda_{\max}$, that is
$$\Delta_i = -\frac{\ln U_i}{\lambda_{\max}}. \tag{4.31}$$
4. We generate a new random number V_i
$$V_i \sim \text{Unif}\big(0, 1\big) \tag{4.32}$$
5. If now the following condition is fulfilled
$$V_i \leq \frac{\lambda(t_{i-1} + \Delta_i)}{\lambda_{\max}} \tag{4.33}$$
we set
$$t_i = t_{i-1} + \Delta_i \tag{4.34}$$

and update

$$i \to i + 1 \tag{4.35}$$

6. Then we go back to step 2. and repeat this until $t_{i+1} > T$. (If the condition in step 5 is not fulfilled, we still go back to step 2 but without updating i and t_i.)

This concludes the algorithm for generating inhomogeneous Poissonian spike trains.

4.3.3 Strength of LGN and background signal

As the rate profile that is output from `pyLGN` is zero-centered and therefore contains negative values, which we do not want (as negative firing rates do not make much sense), we have to make a decision on how to shift and rescale it. We could shift it so that the minimum possible value is 0, but the problem is that the order of the image sequence affect the minimum and maximum value of the signal. Since there are $10! \approx 3.6$ million potential permutations, we cannot simply try all combination to find the minimum possible value. What we did was to simply run a large number of `pyLGN` simulations, where each used a randomized permutation, and then we calculated the minimum value of the output signal for all the simulations. This value was then used to shift the output signal for all new simulations. If any negative values ever occur, they are simply clamped up to zero. The rate profile is also heavily affected by the chosen spatial resolution parameters used in `pyLGN`. We can therefore not simply interprate this directly as the actual firing rate, and only the shape of the signal is viewed as important.

The strength of external input is another set of variables that we need to make a choice with respect to. A loose constraint that we will apply, is that we want the neurons to have an average firing rate on the same scale as observed experimentally in rats, which is around 7 Hz [22]. As the cortical firing rates are to a large degree dependent on the external input, we can calibrate the strength of the stimuli so that the average firing rates corresponds to this. As in [1], we want to use the ratio of Poissonian rate to the threshold rate as reference point for the input strength, as described in section 2.3.1. However, since we have two different external inputs – from both the LGN and the background – we define one such ratio for each of the two, and their sum will represent the total external stimuli. In other words we define

$$\eta_{\text{tot}} = \eta_{\text{LGN}} + \eta_{\text{BG}} \tag{4.36}$$

$$= \frac{\nu_{\text{LGN}}}{\nu_{\text{LGN,thr}}} + \frac{\nu_{\text{BG}}}{\nu_{\text{BG,thr}}} \tag{4.37}$$

where $\nu_{\text{LGN}} = \frac{1}{T}\int_0^T \lambda_{\text{LGN}}(t)\ dt$ is the average LGN firing rate, and ν_{BG} is the constant background rate. The denominators are the corresponding threshold rates as found from equation 2.6.

With some test simulations using sinusoidal LGN signals (see section 6.1), we found that $\eta_{\text{tot}} = 1.1$ gave a satisfactory average cortical firing rate at around 9 Hz (see figure 6.3). The ratio of LGN rate to background rate is then another parameter that we can vary, which will very obviously affect how easily the AI will be able to classify the LFPs.

In Brunel's article, the parameter values of a point neuron network defines a unique state, classified into four distinct domains (see figure 2A in [1]). When we want to vary the amplitude of the LGN signal but want to keep the network in the same state, we can do this by using a background rate with the value

$$\begin{aligned} \nu_{\text{BG}} &= \eta_{\text{BG}}\nu_{\text{BG,thr}} \\ &= \big(1.1 - \eta_{\text{LGN}}\big)\nu_{\text{BG,thr}}. \end{aligned} \tag{4.38}$$

If we increase the LGN rate, we can use equation 4.38 to decrease the background rate in order to keep η_{tot} constant. This will then maintain the network in the chosen state on average. For $\eta_{\text{tot}} = 1.1$ will be in the asynchronus irregular domain, slightly above the border to the synchronus irregular state, as our choice of g value is 5.2.

For the main simulations in this project (section 6.4), the background signal was about 17 times stronger than the LGN signal, a value which was chosen rather arbitrarily.

4.3.4 Poissonian spikes in NEST

`NEST` comes with a built-in inhomogeneous Poisson generator which is easily implemented using `"inhomogeneous_poisson_generator"` as the argument to the `nest.Create()` function. With the `GetStatus()` function we can then send the time steps and values of the rate profile that we get from `pyLGN` into the Poisson generator. It will now send Poissonian spikes according to $\lambda_{\text{LGN}}(t)$ to every neuron it connects to.

Since we want all the synapses of each individual LGN neuron to fire the exact same spike trains, we cannot simply connect the spike generator directly to the cortical populations. This is because that would generate a unique set of spike times for each synapse. We will instead have to generate a population consisting of `NEST`'s `parrot_neuron` models. These neurons work by simply outputting a copy of all incomming spike trains at their synapses. By setting up N_{LGN} parrot neurons and connecting the Poisson generator to these, we can then connect these to the cortical neurons. This will then make sure that every single synapse from

a given LGN (parrot) neuron, outputs the exact same spike trains.

4.4 Part 3 – Spikes to LFP with hybridLFPy

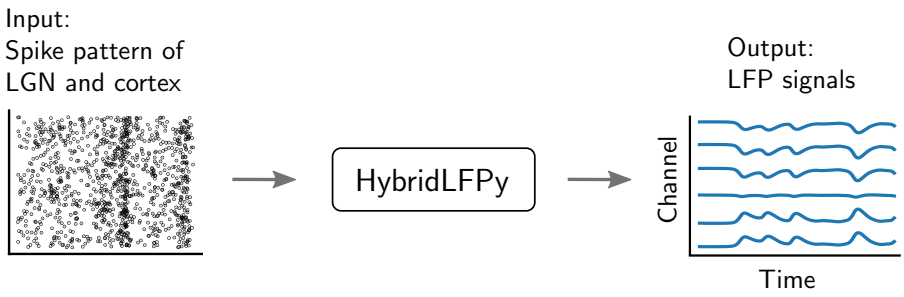


Figure 4.8: Part 3 of the module. The LFP signals are illustrated as being measured at 6 different channels.

The final module we use is the Python package `hybridLFPy` [16], which is an extension of the `LFPy` [25] module. `LFPy` [25] is designed for calculating extracellular potentials from multicompartment neuron models that are built in the `NEURON` simulation environment [18]. LFP calculation requires detailed neuron models and is therefore very computationally expensive to do for large networks. The idea behind `hybridLFPy` is to speed up the calculation by using a pre-simulated spiking pattern from a point neuron network, which can be simulated much, much quicker.

More specifically, by assuming that a network of multicompartment neurons will express the exact same spiking pattern as a point neuron network with the same architecture, we can calculate the spiking pattern beforehand and then "insert" it into a multicomparment network. In other words, we let each multicompartment neuron receive incomming spikes on their dendrites according to the point neuron spiking activity. But whatever happens in the neuron after receiving the spikes, has no effect on the spikes that it itself outputs to other neurons – as the spikes are already calculated. Each multicompartment neuron is therefore only used for generating the LFP, but not for actually generating action potentials. The consequence of this is that we can calculate the LFPs from each of the neurons completely independent of each other – it becomes what one refers to as "embarrassingly parallel".

4.4.1 Further LFP simplification using population rates

Even though `hybridLFPy` drastically speeds up the computation of the LFP, this part is still very much the bottleneck of the simulation. Running part 1 and

2 with a network of of 12,500 point neurons for 1000 ms takes about a minute, but part 3 alone takes around 10 hours. The computation time is in other words almost solely spent on estimating the LFP and not on actually simulating the spiking activity. This motivates a further simplification of the LFP computation, which we describe here.

The objective is to approximate the LFP based on the average **population level** firing rate. This means that instead of basing the LFP calculation on each individual spike from each individual synapse, we instead choose some time interval where we count the firing rates of each population and then estimate the LFP from this.

As described in [16], the LFP fom population u, measured at channel c, is approximated through a temporal convolution

$$\Phi_{cu}(\boldsymbol{r}, t) = (\mathcal{K}_{cu} * \mathcal{U}_u)(\boldsymbol{r}, t), \tag{4.39}$$

where K_{cu} is a kernel and U_u is a vector with the instantaneous firing rates for population u throughout the simulation. The kernels, shown in figure 4.9 for all populations for all 6 channels, are computed as the total average LFP contribution from a single time interval where every single neuron in the population fires exactly once, measured at the given channel.

This approximation scheme reduces the LFP calculation time down from 15 hours to the order of about a second.

Adding induced LFP into the kernel

Since both cortical populations will induce neural activity in each other, we need to consider this when calculating the kernels. In other words, since both populations are connected to each other, when the one population fires, it will force neural activity in the other cortical population which will have some contribution to the LFP. This small contribution must be added into the kernel for each population.

4.4.2 Note about synapse models

In the point neuron network we used delta synapses for efficiency reasons (see section 2.3.1). But since we use alpha synapses in the the `hybridLFPy` part of the simulation (see section 2.3.2) for more realism, we need to make sure that the synapse strengths are calibrated so that the synapses transfer the same total amount of current for each activation. The transmembranial current for the two

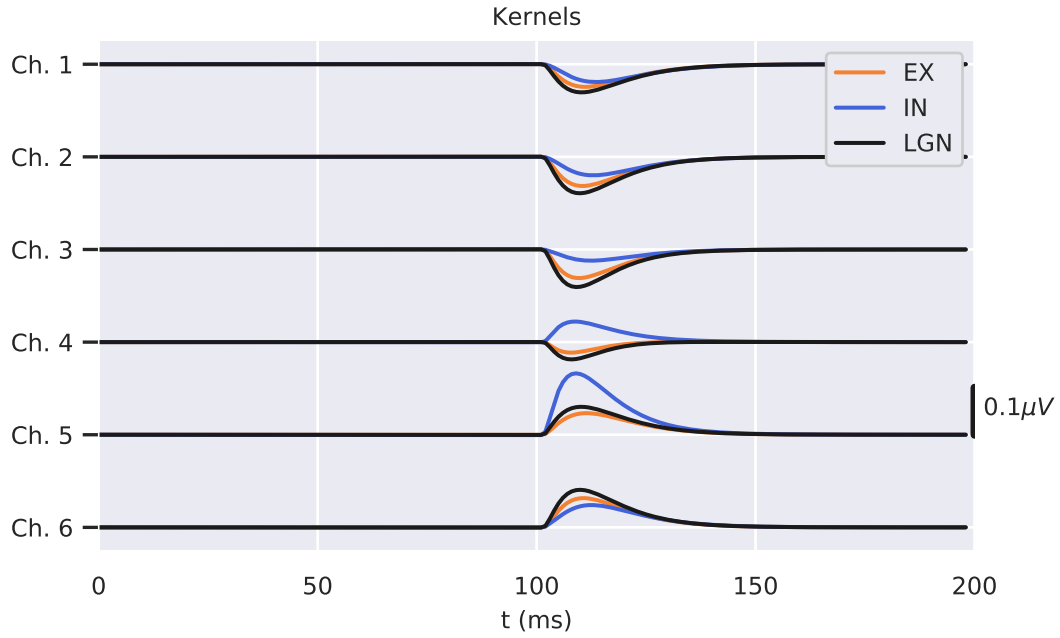


Figure 4.9: The kernels $\mathcal{K}_{cu}$ for all 6 channels for all three populations. This particular kernel is created from a network with parameters $N_{\text{LGN}} = 1000$, $N_{\text{EX}} = 10,000$ and $N_{\text{IN}} = 2500$, $g = 4.5$ and $J_{\text{EX}} = 0.1$.

models for a synapse that activates at time $t = 0$ is written as

$$I_{\text{delta}}(t) = J_{\text{delta}}\delta(t) \tag{4.40}$$

and

$$I_{\text{alpha}}(t) = J_{\text{alpha}} t e^{1-t/\tau_{\text{syn}}}, \tag{4.41}$$

respectively.

To make sure that the delta and alpha synapses induces the same amount of current per spike, we scale the value of J_{alpha}. In other words, the following relation must be fulfilled

$$J_{\text{alpha}} \int_0^\infty t e^{1-t/\tau_{\text{syn}}}\, dt = J_{\text{delta}} \int_0^\infty \delta(t)\, dt \tag{4.42}$$

where the right integral is of course equal to 1 and the left integral is equal to $e\tau_{\text{syn}}^2$, giving the relation

$$\boxed{J_{\text{alpha}} = \frac{J_{\text{delta}}}{e\tau_{\text{syn}}^2}}. \tag{4.43}$$

4.5 Running the simulation

Each simulation is run with a unique number – a simulation index – which will be used to generate unique seeds for its random number generators. The output for each simulation is an array that contain the LFP signals for each of the channels, of which we will have 6 in this project. The LFPs are then stored in files, where the filenames contain the simulation index in addition to the order of the image sequence used as input. The last part is crucial, as we need this to keep track of the labels for the classes.

- All the scripts that compose this simulation are found at the following Github repository: `https://github.com/Kodemannen/Brunel-with-optical-input/tree/master/simulation`
- The `PyLGN` script that we used for part 1: `LGNsimulation.py`
- The `NEST` script used for part 2: `nest_simulation.py`
- The `hybridLFPy` script and other files used to create the LFP approximation kernel described in part 3 is found in the subfolder `/kernel_creation`. The subfolder `/morphologies` contain `.hoc` files give `hybridLFPy` the morphologies for the neurons. These were chosen simply because some were needed, and no real consideration were given to the choice of these.
- Simulations were run on the Abel computer cluster `https://www.uio.no/english/services/it/research/hpc/abel/more/index.html` using SLURM Workload Manager. Running 100k simulations distributed over 200 jobs with a network of 12,500 neurons for 3000 ms with a time resolution of 0.1 ms took approximately 15 hours.

The parameter values used in the simulations are shown in table 4.1 below.

Parameter	Value	Parameter description
N_{EX}	10,000	Excitatory cortical population size
N_{IN}	2500	Inhibitory cortical population size
N_{LGN}	1000	LGN population size (parrot neurons)
J_{EX}	0.1 mV	Excitatory delta synapse strenght
J_{IN}	$-gJ_{\mathrm{EX}}$	Inhibitory delta synapse strenght
g	5.2	Ratio of inhibitory to excitatory synapse strength
ϵ	0.1	Connection probability
C_{m}	1 pF	Membrane capacitance
V_0	0 mV	Resting potential
V_{thr}	20 mV	Threshold potential
V_{reset}	10 mV	Reset potential
τ_{m}	20 ms	Membrane time constant
τ_{syn}	5 ms	Synaptic time constant (for alpha synapses only)
τ_{ref}	2 ms	Refractory period
τ_{delay}	1.5 ms	Axonal delay
n_{channels}	6	Number of electrode recording channels
η_{tot}	1.1	Representing the size of the total external input (see section 4.3.3)

Table 4.1: Parameters used in the simulation. Note that J_{EX} were also used as synapse strength for the LGN population and the background noise, in addition to the excitatory cortical population.

Chapter 5

Methods II – Classifying the LFP using artificial neural networks

In this section we go through how we preprocess the LFP data, and how we employ the deep learning framework `Keras` to build the artificial neural networks (ANNs) that we use for classification.

5.1 Data preprocessing

Much of the work involved with deep learning goes into preparing the data for training. Different preprocessing methods suit different problems, so making good decisions here is important.

The raw LFP data are matrices of dimension 6×3000, being, respectively, the number of channels and time steps. We begin by cutting off the parts corresponding to where the grey image was shown, meaning the first and last 250 ms. After this, each datapoint is an array of dimension 6×2500. For each datapoint we also create a vector of integers that represents the image sequence that was used as input for that particular simulation. These will of course be used to keep track of the labels for the data.

After this we stochastically distribute the data and label arrays into three separate sets – a training set, a validation set and a test set. As is commonly done, 60% of the data is put into the first one, while the remaining two will get 20% each.

Next we calculate the per channel mean and standard deviation of the training set. These values are then used to zero-mean and normalize all three datasets. In other words, for each channel in each datapoint in each set, we subtract the training set mean and divide by the training set std. This ensures that the mean and std of each input data is around 0 and 1, which is conventional to use together

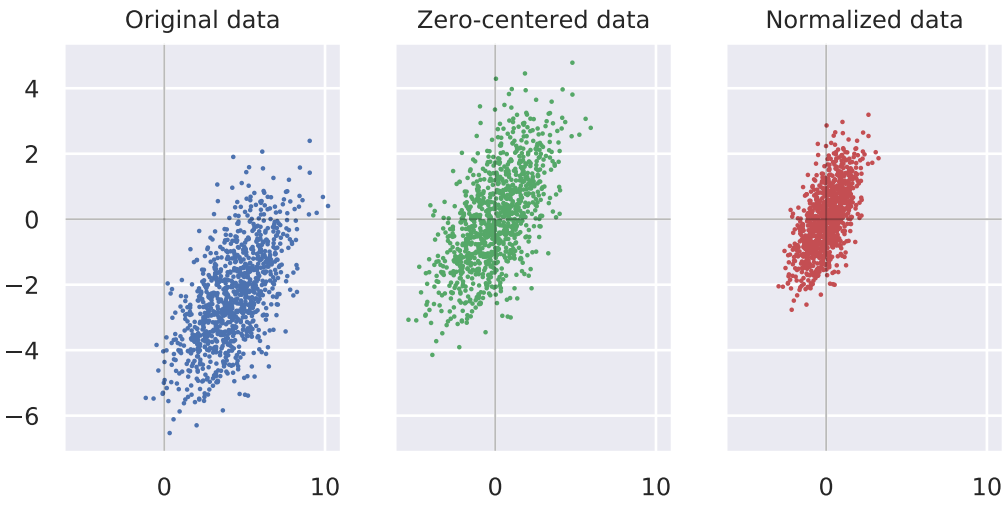


Figure 5.1: Toy data used to illustrate zero-centering and normalizing.

with Batch Normalization [21] for optimizing the forward and backward flow of information in deep neural networks. These operations are illustrated on some 2-dimensional toy data in figure 5.1.

The reason we draw these values from the training set only, is because this is a better test for generalization. For example, if we would want to predict the vision of a biological brain in real time (a concievable extension of this project), we would not have prior access to the mean and std of the data, and the input signal would therefore have to be preprocessed in real time using averages and stds from the training set only.

Moving on, we cut each datapoint and corresponding label sequence into 10 pieces. Each datapoint is now a 6×250 matrix and each label a single integer between 0 and 9. This of course mean that we have 10 times the datapoints that are 1/10th the original size. We then shuffle all the datasets. The data and labels are of course shuffled in the exact same way so that datapoint at index j has the label at index j.

Finally the data is smoothed with an order 8 Chebyshev type I lowpass filter and downsampled with a factor of 5, using the function `signal.decimate` found in the `SciPy` library [12]. This reduces the dimension of the data which will drastically cut down the number of parameters needed in the artificial neural networks. After this the datapoints are matrices of dimension 6×50 and are ready for training Figure 5.2 shows an example of a datapoint before and after downsampling.

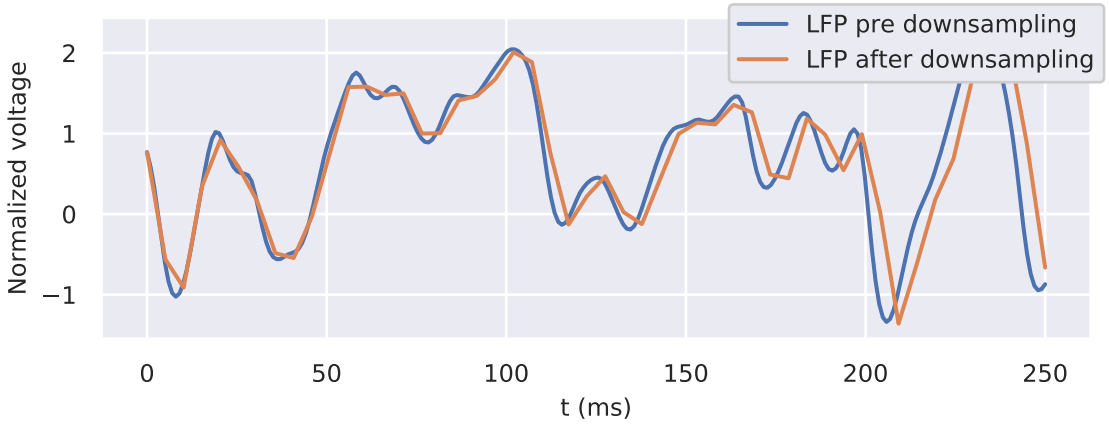


Figure 5.2: A datapoint before and after downsampling.

5.2 Building and training ANNs with Keras

Building computationally effective artificial neural networks have never been easier, with an abundance of different open-source libraries to choose from. A very popular choice is the Python package `Keras` [4] with a `TensorFlow` [11] backend, which is what we use here. All that is needed is to import the datasets to memory, specify the architecture and hyperparameters of the ANN, choose some optimizing scheme and initiate the training.

The types of networks we use here are a DNN, a CNN and an LSTM. All of these are simple to set up and run in `Keras`, and as the data is quite lightweight after downsampling, the training can be run on a regular home computer in only a few hours or less.

During training, we stochastically sample a small subset – a mini-batch – of data from the training set (without replacement), perform the forward pass, calculate the gradient and update the weights. Then we sample a new mini-batch and repeat this process over and over until we have gone through the whole training set. An **epoch** is defined one full cycle through all the training data, and after each epoch we evaluate the network on the validation set (see section 5.2.3)

Here we briefly go through most of the techniques we use for the training (except for two, which which will just refer to, which are Dropout [36] and MaxPooling [32]). Roughly the same training scheme is used for all the three networks, but we vary the hyperparameters somewhat.

5.2.1 The Adam optimizer

After some some trial and error on data from some test simulations, we decided upon the Adaptive Moment Estimation (Adam) algorithm [23] for our choice of optimizer. It is a stochastic, gradient-based optimization technique that has become very popular because of its effectiveness and memory efficiency. The algorithm makes use of the momentum – which is the accumulation of gradients over time, analogous to the build up of physical momentum of an accelerated body – to effectively adjust the learning rate over time (i.e. training cycles).

In the end, Adam is just one of many methods to find a vector to move in parameter space in order to optimize the loss function $\mathcal{L}(\boldsymbol{X}, \boldsymbol{Y}|\boldsymbol{\theta})$, and we refer to [23] for more details on precicely what separates it from other gradient based optimizers. After the gradient is found, the update of the parameters is done exactly as in equation 3.10, 3.11 and 3.12.

The Adam optimizer is found in the `Keras.optimizers` module.

5.2.2 Batch normalization

An operation that has become standard in deep learning is Batch normalization [21]. It is a method designed to increase the information flow forward and backward in the network. Before its invention, one would struggle with vanishing activations in very deep neural nets, meaning that it would often happen that the deepest layers would receive only zero vectors as input. What batch normalization does is to shift and normalize the activations $\boldsymbol{a}^l$ (see equations 3.7 through 3.9) of every layer for each forward pass, according to the equation

$$\hat{a}_j^l = \frac{a_j^l - \mathbb{E}(a_j^l)}{\sqrt{\text{var}[a_j^l] + \xi}} \gamma^l + \beta^l. \tag{5.1}$$

The expectation value and variance are computed over the mini batch of data, and ξ is a tiny constant to guard against zero-denominators. In other words, when we do a forward pass and calculate the activations of a layer $\boldsymbol{a}^l$, we calculate the elements in vector $\hat{\boldsymbol{a}}^l$ according to equation 5.1, and replace the original activation vector with this. The parameters γ^l and β^l are trainable, and will be updated with gradient descent according to the equations in section 3 in [21]. At test time, the expectation values and variances that we insert into equation 5.1 will be taken from the full training set. The values for γ^l and β^l will be whichever values they reached during training.

An interesting side effect of this operation is that, during training, since the activations are shifted and normalized over the mini batches, the output for a given datapoint is non-deterministic. The reason is that the values used for

shifting and normalization is dependent on the specific batch of data, which is randomly selected. This turns out to have an additional regularizing effect that actually helps against overfitting (see next section) [21].

A class for implementing Batch Normalization is found in the `keras.layers` library. Note that it is not used with the LSTM.

5.2.3 Preventing overfitting

When performing function fitting, which is what machine learning essentially boils down to, problems arise when the function is tuned too well to the training data. This is known as **overfitting** and is something that will make the network perform worse on new data. Combating this is an important area of machine learning research, and many techniques have been developed and have become standard practice. We have here applied two common procedures.

L2 regularization

The first procedure is known as L2 regularization. This involves adding a penalty for weight matrices with a high L2 norm (defined as the sum of the square of the matrix elements) into the loss function. In the case of our cross entropy loss (equation 3.14), this is written as

$$\mathcal{L}(\boldsymbol{X}, \boldsymbol{Y}|\boldsymbol{\theta}) = -\frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{c} y_j^i \log \hat{y}_j^i + \frac{1}{2}\rho \sum_{l=1}^{L} |\boldsymbol{W}^l|^2, \tag{5.2}$$

where ρ is a hyperparameter that decides the strength of the regularization, and L the number of layers in the network (excluding the input layer). When we now calculate the gradient of $\mathcal{L}$, we will of course have to include the gradient of the L2 term as well. This easily found as

$$\frac{\partial}{\partial w_{mn}^k}\left(\frac{1}{2}\rho \sum_{l=1}^{L} |\boldsymbol{W}^l|^2\right) = \rho w_{mn}^k. \tag{5.3}$$

We see now that the factor $\frac{1}{2}$ in equation 5.2 is only there to make the gradient look cleaner.

Earlystopping

This last procedure is the very reason that we create a validation set in addition to the training and test sets. The idea is simple: After each epoch, we check how

well the network performs on data that it have not been trained on – i.e. the validation set. The accuracy of the predictions on this set will indicate how well the network generalizes to new unseen data. If the loss and/or accuracy on the validation set suddenly start increasing at some point after some epochs, then that is an indication that the network has begun to overfit and we can terminate the training.

However, since the loss during training often increases and decreases locally many times during the full training period, we must decide on some parameter that defines how patient we are in deciding when it looks like overfitting. In other words we keep track of the best loss and/or accuracy on the validation set, and if the loss has not improved after some number n_{patience} of epochs, then we say that the network has started overfitting and can terminate the training. We then revert back to the configuration that gave the best score, which we have stored.

Earlystopping is found in the `keras.callbacks` module.

Chapter 6

Results

In this chapter we go through the results of the experiments that we performed. We executed two phases of test simulations in order to calibrate and benchmark the simulation, and we include the analysis from these.

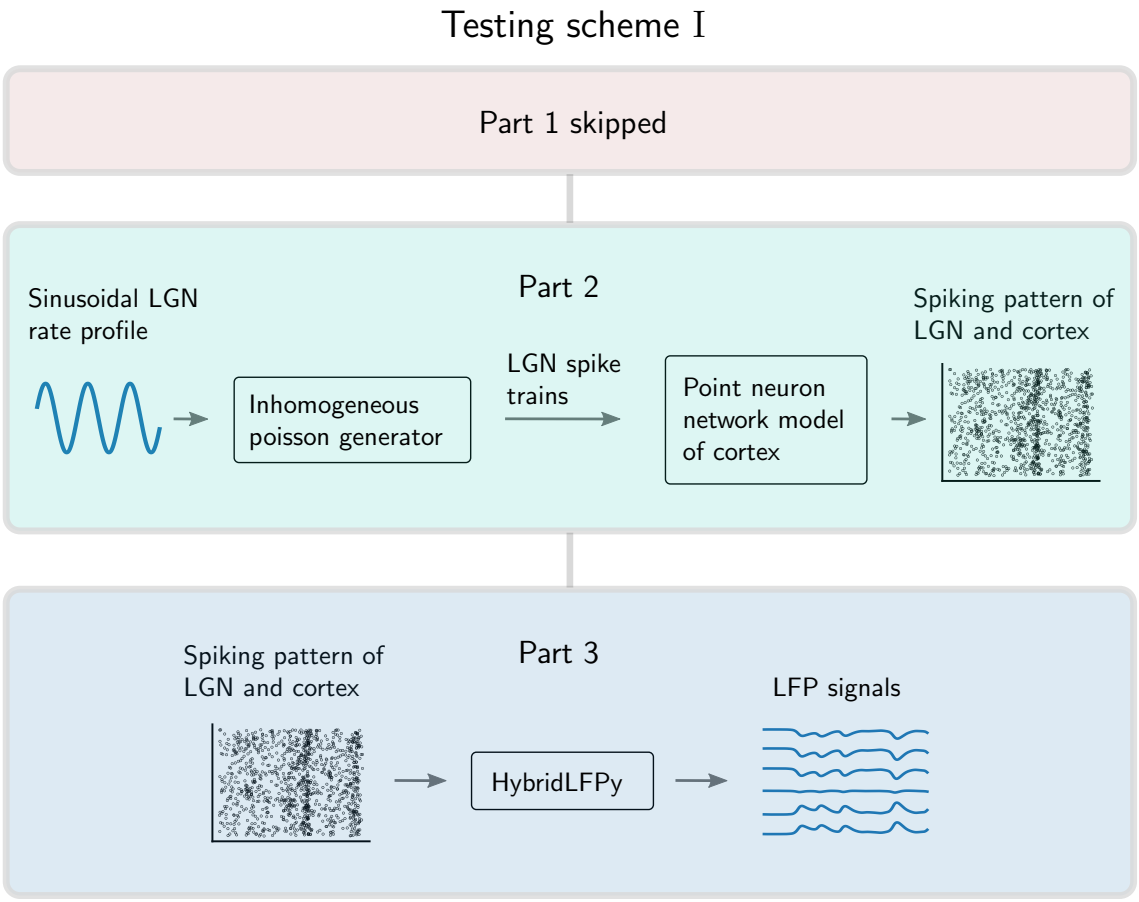


Figure 6.1: Scheme for the first phase of testing with sinusoidal stimuli.

6.1 Test phase I – Sinusoidal input

We begin by skipping part 1 of the simulation and instead create an artificially simple LGN rate profile. This is done so that we can carefully analyze how the cortical model reacts to very simple stimuli. By letting the LGN firing rate oscillate sinusoidally with a given frequency f_λ, we began by seeing if we could re-extract the frequency from the measured LFP. In other words, the LGN firing rate profile used here is

$$\lambda_{\text{LGN}}(t) = A_\lambda \sin(2\pi f_\lambda t) + b_\lambda, \quad 0 \leq t \leq T, \tag{6.1}$$

where the simulation time T was chosen to be 1001 ms. Since we do not want negative rates, we choose $b_\lambda = A_\lambda$ so that λ_{LGN} oscillates between 0 and $2A_\lambda$ with a mean of A_λ. We use this to inspect how the cortical network reacts to various simple signals by varying f_λ and A_λ between simulations. Figure 6.1 shows an illustration of the test scheme.

Note that both λ_{LGN} and f_λ are in units of Hz. In other words, we have two rates: the rate λ_{LGN} at which the LGN population fires, and then we have the rate f_λ at which λ_{LGN} itself oscillates.

Varying the input frequency

We start by only varying the frequency f_λ of the sinusoidal input. The amplitude is fixed to $A_\lambda = 3$ Hz, which gives us an η_{tot} that oscillates between 1.07 and 1.13 with a 1.1 average. As mentioned in section 4.3.3, this value of η_{tot} was rather arbitrarily chosen, with the simple constraint that we want the average cortical firing rate to be in the same order of magnitude as observed in biological brains. In figure 6.3 we have plotted the firing rates averaged across multiple simulations. The rate oscillates around 9.3 Hz for both cortical populations, which we deemed sufficiently close to experimental observations.

The lower limit of f_λ will be bound by T, as it makes no sense to use a signal without at least one or more periods within the simulation, and the upper limit will be somewhat bound by the time resolution as well as the maximum rate that the network in practice will be sensitive to (assuming there is one, which seems likely). We therefore began by choosing 6 different input frequencies $f_\lambda = 4$, 10, 25, 45, 70, and 110 Hz for a simulation time $T = 1001$ ms.

Figure 6.2 shows measures of network activity for $f_\lambda = 4$ Hz and $f_\lambda = 45$ Hz. From subplots G and H, is clearly not immediately obvious which of the LFPs that belong to which input, just from looking at it. One thing to mention is that since the scale of the y-axis in subplot C and D is over ten times larger than the scale of the time-axis, the rasters are squeezed vertically and the spiking activity

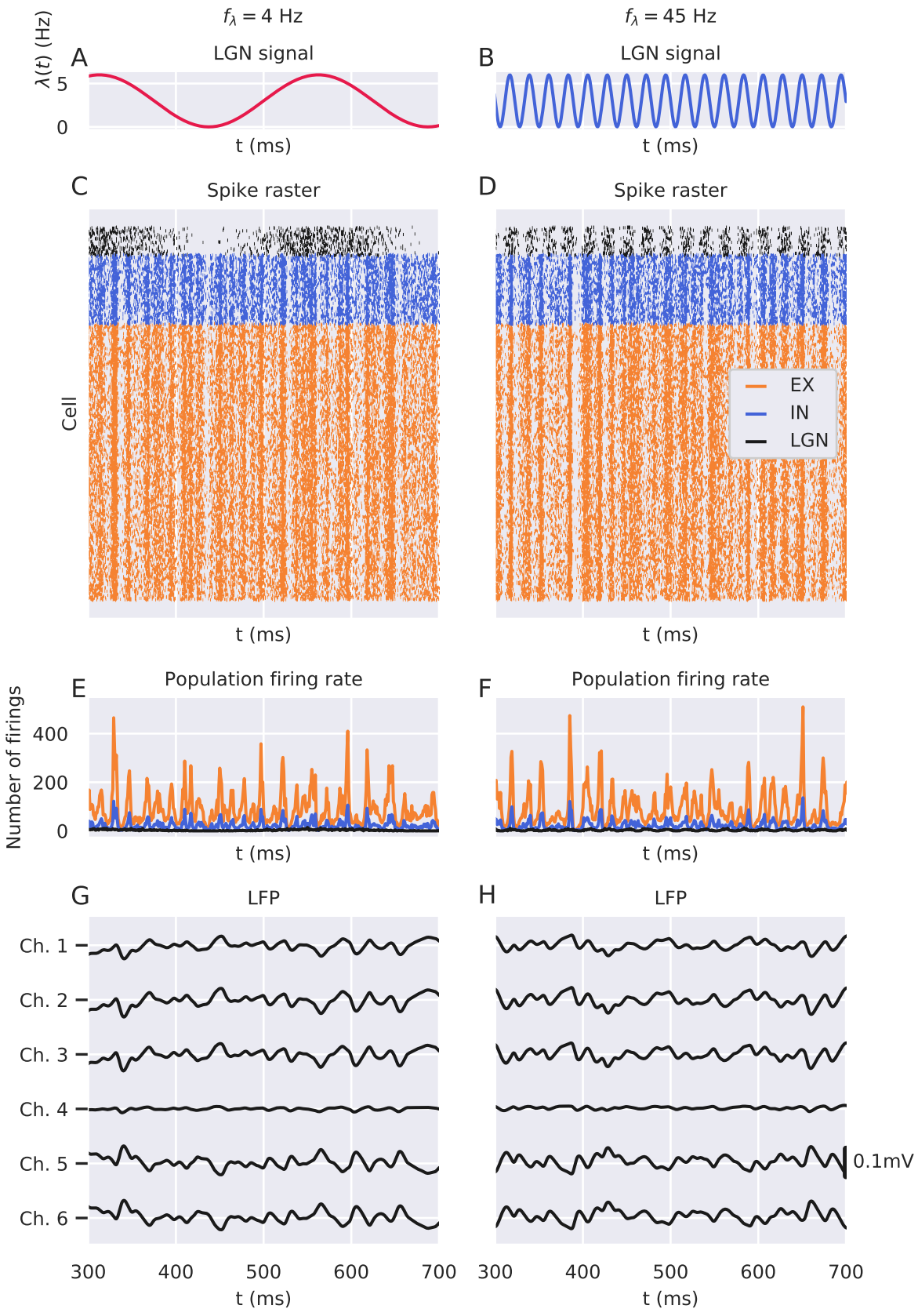


Figure 6.2: Measures of the network activity for two different input signals from LGN. Each column represents one simulation. Note that the time axis is truncated.

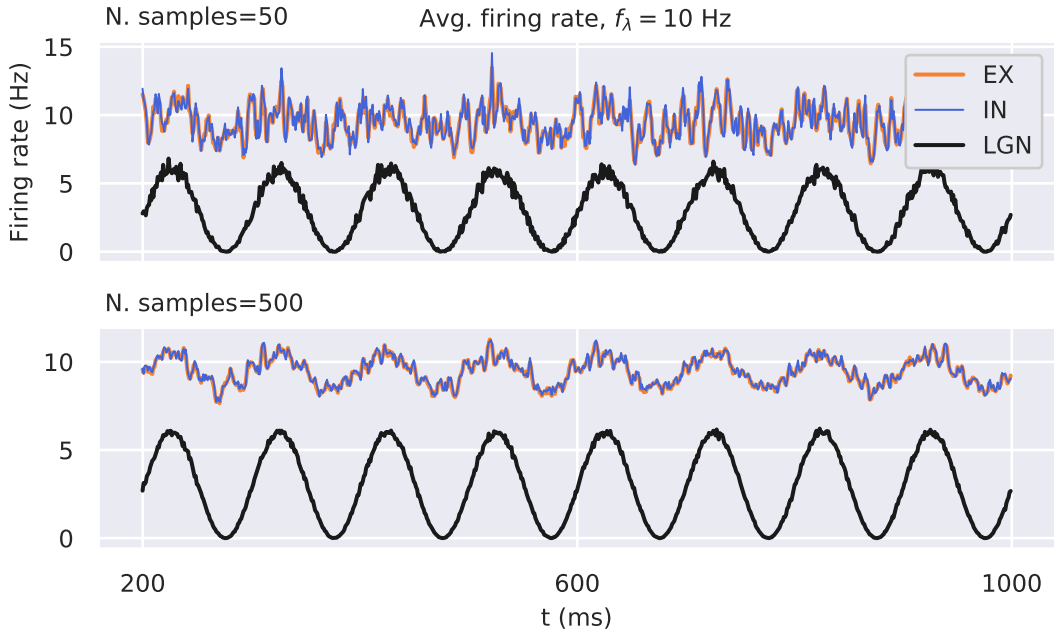


Figure 6.3: Neural firing rate over time for each of the three populations, averaged over multiple simulations, with input frequency $f_\lambda = 10$ Hz.

look more synchronous than it really is.

PSD analysis of LFP and population rates

To make sure that the information about the input signal is actually to be found in the LFP, we did a power-spectral-density (PSD) analysis [39] of the channel 1 LFP and averaged it across 500 simulations per value of f_λ. The result is shown above in figure 6.4. The colors mark frequency spectra for simulations with different input signals. As we see, there are clear peaks at $f_\lambda = 4$, 10, 25, 45, 70, and 110 Hz, which were the input frequencies. This is precisely what we would hope for.

However, since some of the LFP is a direct contribution of the LGN synapses, which would very obviously contain the input frequencies, we want to ensure that we can find the same peaks in the cortical contribution to the LFP as well. This was easily done by doing a similar analysis as explained above, but removing the direct LFP contribution from the LGN synapses (equivalent to setting the LGN part of the kernel [fig. 4.9] to zero). The result of this are the dashed, black lines in the figure. Here we did not color encode distinct frequencies.

Figure 6.4 shows that the information about the input signal is clearly to be found in the LFP, but that it decreases with higher frequency. The amount that

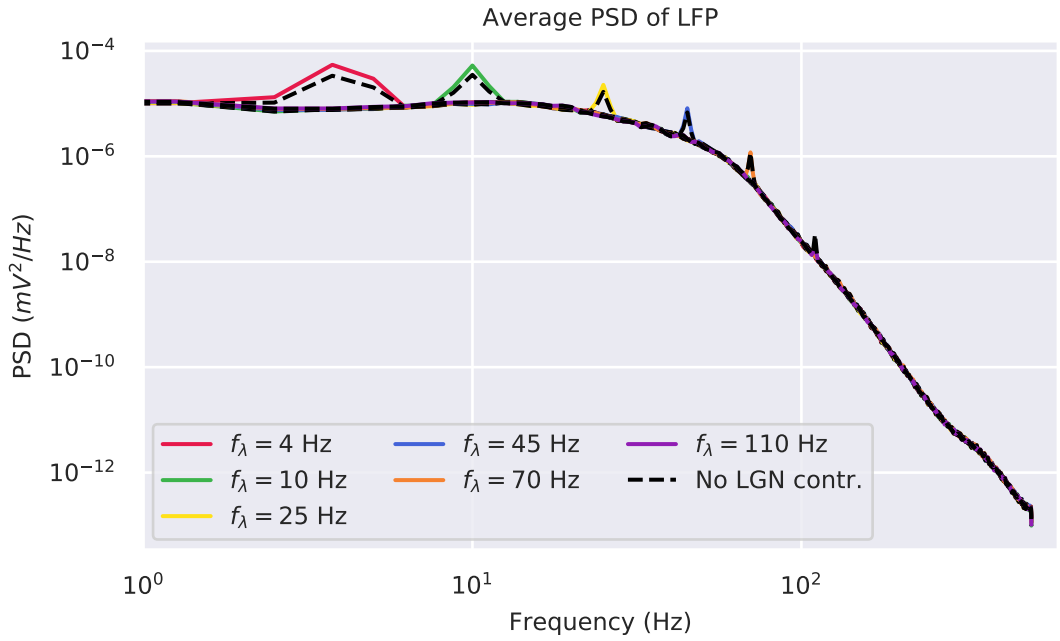


Figure 6.4: The coloured lines are the average PSD of the LFP of channel 1 for each value of f_λ, averaged across 500 simulations. The black, dashed lines represent the same as the coloured, but where the LFP is calculated without the direct contribution from the LGN synapses.

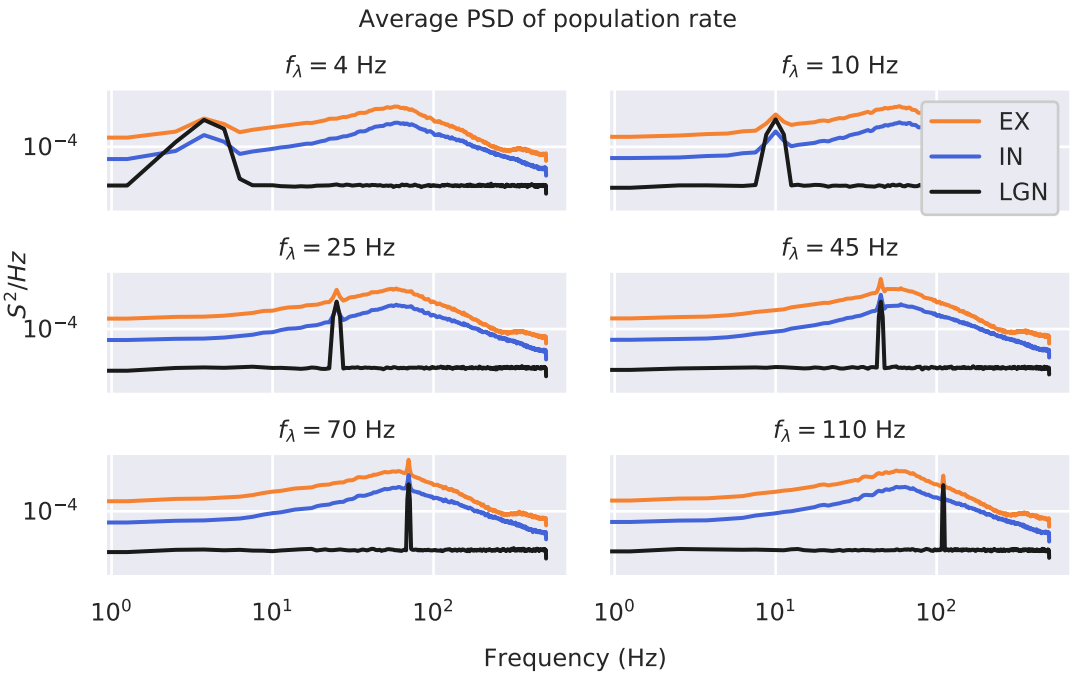


Figure 6.5: PSD analysis of the population firing rates series. This is consistent with fig 8B in Hagen et al [16].

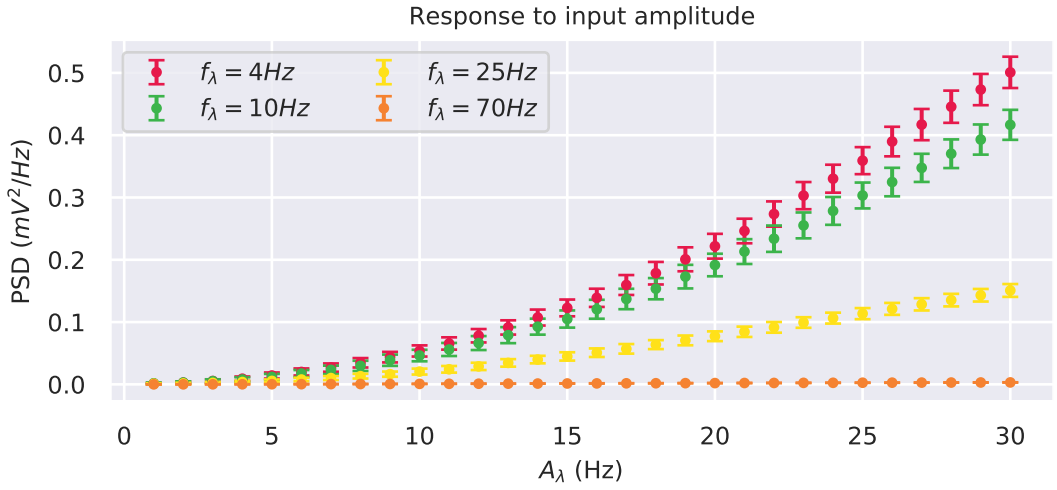


Figure 6.6: Each point is showing the average value of the peak of the PSD for the given input frequency f_λ with the given amplitude A_λ.

was just the direct conribution from the LGN synapses was around 20 % (i.e. the difference between the coloured and dashed black line at a given peak, divided by the amplitude of the coloured peak), but went up a little with higher frequencies.

We also did a PSD analysis of the time series of the population rates, shown in figure 6.5, and compared it to fig 8B in Hagen et al [16]. We see that this is consistent.

Varying the input amplitude

Next we examined how the input information is dependent on the strenght of the input. For four of the frequencies from the previous subsection, namely $f_\lambda = 4, 10, 25$ and 70 Hz, we let the amplitude A_λ vary from 1 to 30 Hz. Since we vary the amplitude and thus the mean of the LGN signal, that means we had to lower the background signal accordinly using equation 4.38, in order to have the network be in the same state (again, see [1]).

NB! Note the background rate was set with the mean of the LGN signal in mind. In other words, ν_{BF} was held constant throughout each simulation.

For each combination of f_λ and A_λ, we ran 500 simulations and computed the average PSD. We then calculated the amplitude of the peaks in the PSD for each value of f_λ. The amplitudes of these peaks are plotted as a function of A_λ in figure 6.6 for each of the four input frequencies. We see from the figure the higher the input amplitude, the more the lower frequencies will dominate.

Result: The frequencies in the input signals can definitely be extracted out again from the LFP. Moreover we see that the effect of the input on the network seem to disappear somewhere after around 100 Hz. It must also be mentioned that even though we see clear peaks for the input frequencies in figure 6.4, this is the average PSD across many simulations, and it is not necessarily the case that the information about the input can be as easily extracted from a single example.

We can infer from figure 6.6 that if an LGN signal contain components with both high and low frequencies, as the input amplitude goes up, the more lowpass filtered the signal will be. This could mean that it is not necessarily the case that increasing the amplitude of the input signal will always make it easier to pick up. If much of the information in the signal is in the high frequency domain, it might potentially make it harder.

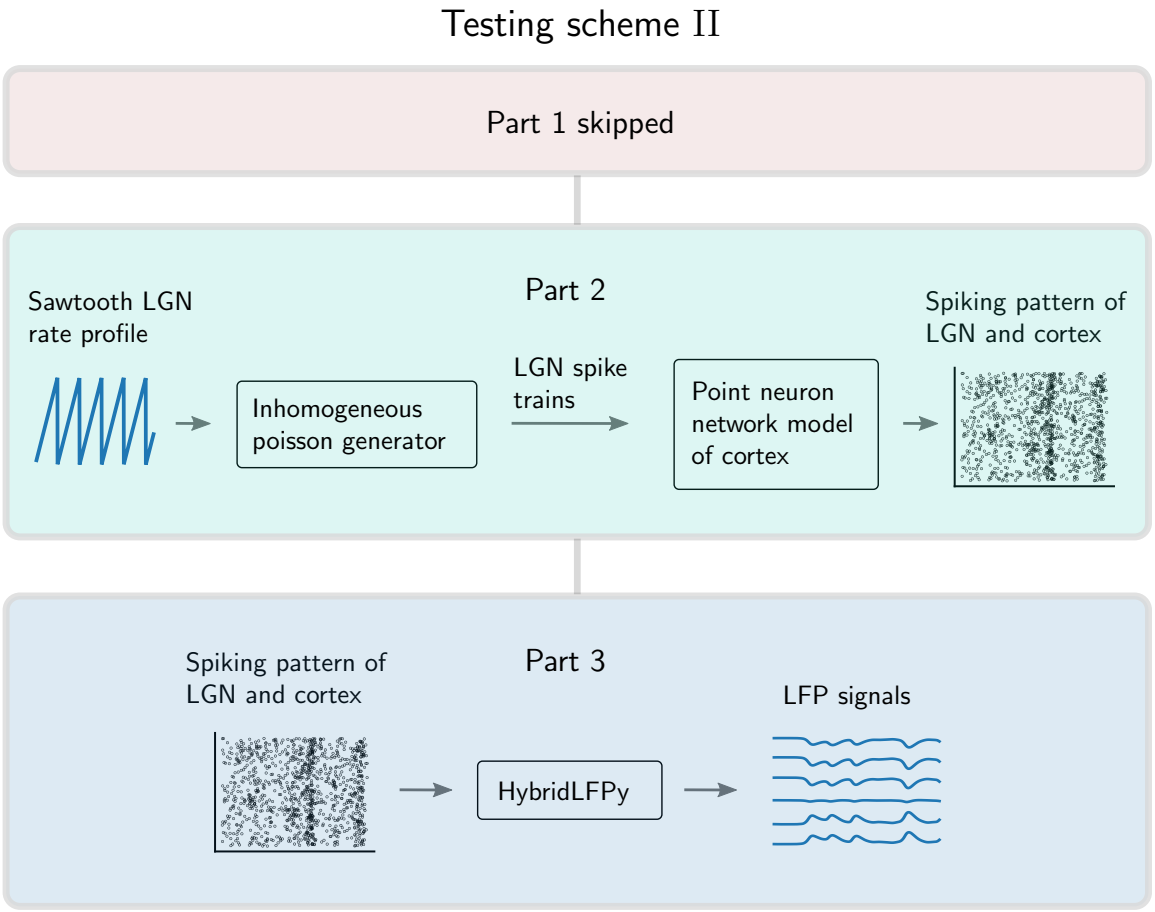


Figure 6.7: Simulation scheme for the second phase of testing where we used sawtoothed LGN stimuli.

6.2 Test phase II – Classifying sawtooth input

As a further sanity check, we begin by making sure that we can classify LFPs generated from simple LGN signals. To do this, we use sawtooth shaped and reverse sawtooth shaped LGN signals using 5 different frequencies. We again choose the values $f_\lambda = 4, 10, 25, 45$ and 70 Hz. This makes a total of 10 different inputs that vary in both shape and frequency (see figure 6.8). In other words, we generate

$$\lambda_{\text{LGN}}(t) = A_\lambda \,\text{sawtooth}(2\pi f_\lambda t) + b_\lambda, \quad 0 \leq t \leq T, \tag{6.2}$$

where again $b_\lambda = A_\lambda$, and $A_\lambda = 3$ Hz.

For each signal we ran 10,000 simulations for a time $T = 1001$ ms, making a total of 100,000 LFP datapoints after cutting them into pieces (see section 5.1). The algorithms used for classification in this section are PCA, DNN and CNN.

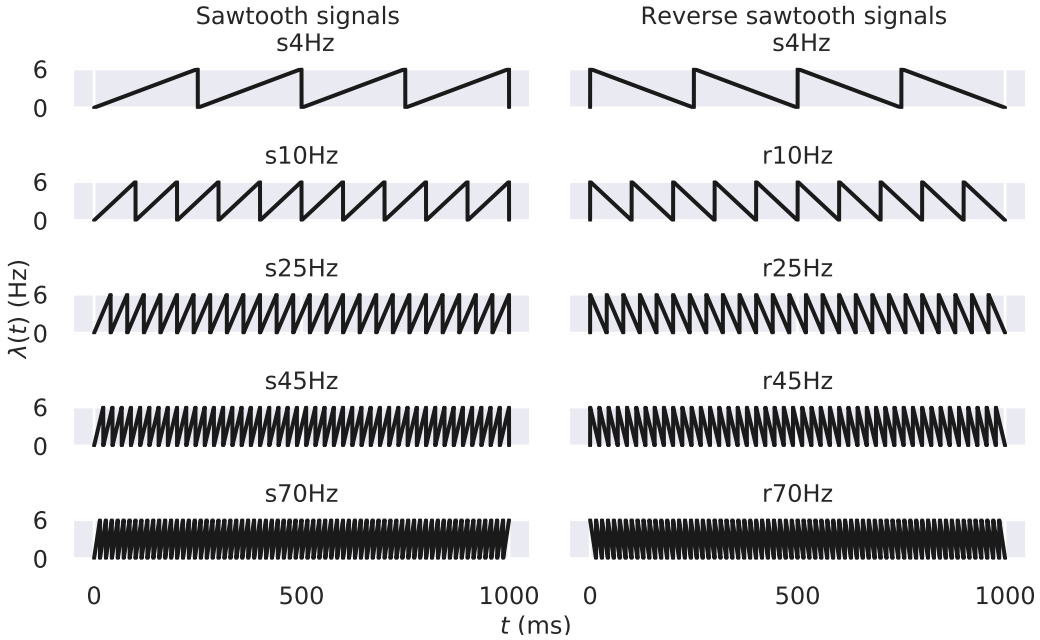


Figure 6.8: 10 different simple sawtooth shaped LGN signals. The s stands for "sawtooth" and the r for "reverse sawtooth".

We present the results in that order, but the figures of the results are shown together at the end of the section.

Each of the 10 signals defines its own class, and the abbreviations used to name the classes in this section contain an "s" or "r", standing for "sawtooth" and "reverse sawtooth", as well as a number that gives frequency of that particular signal. The signals with their corresponding names are shown in figure 6.8.

6.2.1 Classification with PCA

As a benchmark we begin by doing a principal component analysis (PCA) on the dataset. It is linear way to reduce the dimensionality of data, which for sufficiently simple data can be used for classification. For a derivation of the algorithm, see appendix 8.1. If we can classify the sawtooth data to a satistfactory degree with PCA, then there is no point in using slower, more complex algorithms like ANNs.

Result: PCA was not sufficient to classify the data, except for the lowest frequencies. Figure 6.9 shows the data visualized in PC space for the four most dominant axes. Only the three most principal components show any clear separation of classes at all. Unsurprisingly, more complex, non-linear classification algorithms seem to be required.

6.2.2 Classification with DNN

The natural next step is to see how well a dense neural net (see section 3.2) can classify the sawtooth LFP data. It can model non-linearities, but is at the same time very easy to implement and quick to train.

Result: The best DNN configuration reached an average accuracy of 97.9 % on the test set. Its training scheme is found in table 6.1. Like we saw with PCA, the higher frequencies were harder to classify, as seen in the confusion matrix in figure 6.11, which shows the rate at which image i is mistaken for image j. The diagonal shows the true positives, and the mean of this gives the average accuracy. The training development is seen in figure 6.10.

We did not perform any systematic search in hyperparameter space, so better configurations are likely possible to find with e.g. a grid search. The training time for the DNN was in the order of a minute.

Classification scheme	Value
Algorithm	Dense neural net (DNN)
# hidden layers	2
# nodes per layer	966, 128, 64, 10
Activation	ReLu
Output activation	Softmax
Optimizer	Adam
- Learning rate	0.01
- Decay	0.03
L2 regularization	0.0005
Batch size	512
Earlystopping patience	8

Table 6.1: Network and training scheme for the best DNN on the sawtooth stimulated LFP data. The Adam optimizer worked slightly better than SGD with Nesterov momentum. **NB!** Note that the listing of nodes per layer includes the input and output nodes.

6.2.3 Classification with CNN

Even though we reached a very good accuracy using a simple DNN , it is still interesting to see if a convolutional neural net (see section 3.4) can improve the results even further. It is slower to train, but has the property that it can easier take into account smaller, more local patterns in the signal. Our prior expectation is therefore that it should exceed the performance of the DNN.

Result: The CNN managed to reach an accuracy of 100 % for every single class. We did not have to try many different architectures to do this, so the sawtooth data seems to be a rather easy classification task for a CNN. The training scheme used is shown in table 6.2, and training development and confusion matrix in figure 6.12 and 6.13. The training time was about an order of magnitude higher than for the DNN, meaning it took somewhere around 10 minutes.

As this dataset was such an easy task for both the DNN and CNN, there is not much interesting to report on the performances of different network architectures. The configuration we used is shown in table 6.2 below, but many others would give the same results.

Classification scheme	Value
Algorithm	Convolutional neural net (CNN)
# conv. layers	3
Kernel sizes	5, 5, 5
# kernels per layer	32, 64, 128
Strides	1, 2, 2
Maxpool	Yes
Pool size	2
# dense layers	1
# nodes per dense layer	128
Activation	ReLu
Output activation	Softmax
Batch normalization	Yes
Optimizer	Adam
- Learning rate	0.01
- Decay	0.03
L2 regularization	0.0005
Dropout conv. layers	0.75
Dropout dense layers	0.5
Batch size	512
Early stopping patience	4

Table 6.2: Network and training scheme for the best CNN on the sawtooth stimulated LFP data. The network is built with a classical architecture of convolutional blocks followed by dense layers. Each convolutional block consists of a convolutional layer, activation, batch normalization, maxpool and dropout, in that consecutive order.

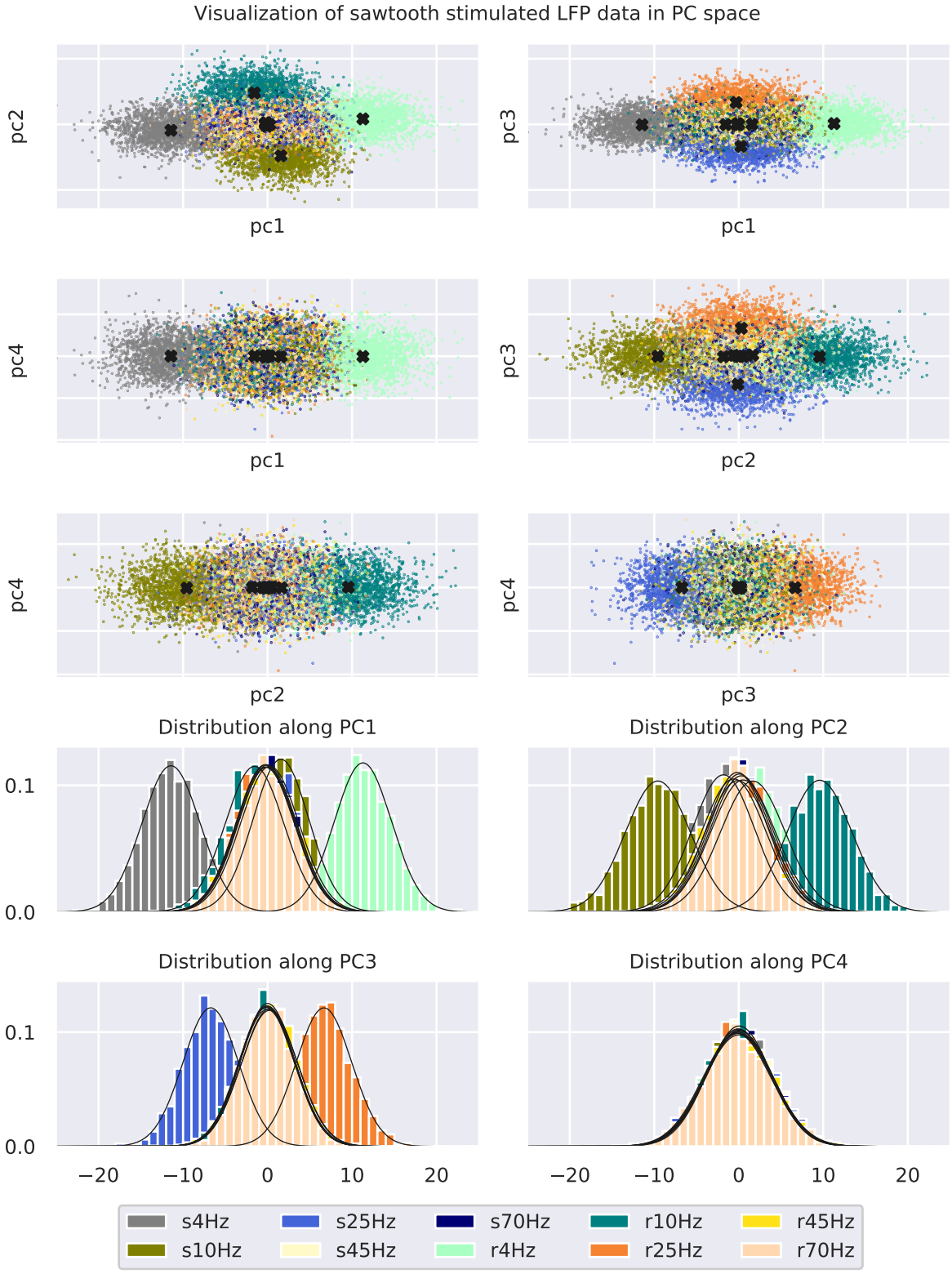


Figure 6.9: Upper 6 boxes: The data on the 4 largest PCs plotted pairwise against each other. Only the low frequency signals can be adequately separated. The black Xs are the centers of mass for each of the clusters. **Lower 4 boxes:** Showing the distribution of the data along the 4 largest PCs. The black curves are fitted Gaussians for each class.

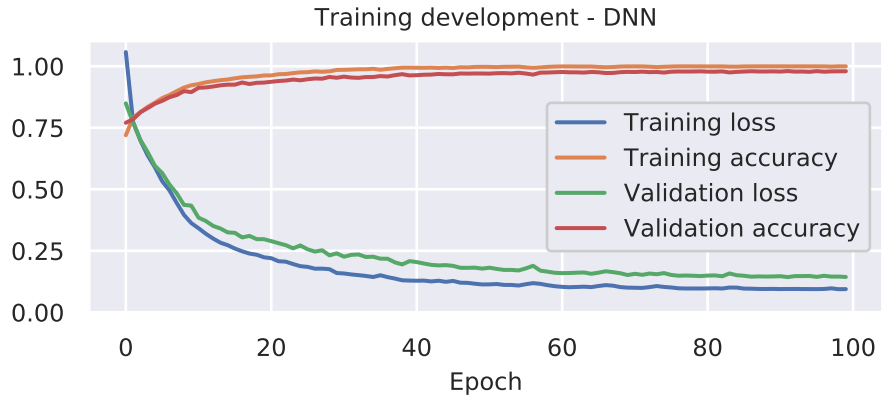


Figure 6.10: Loss/accuracy development during training for the DNN on the sawtooth LFP dataset.

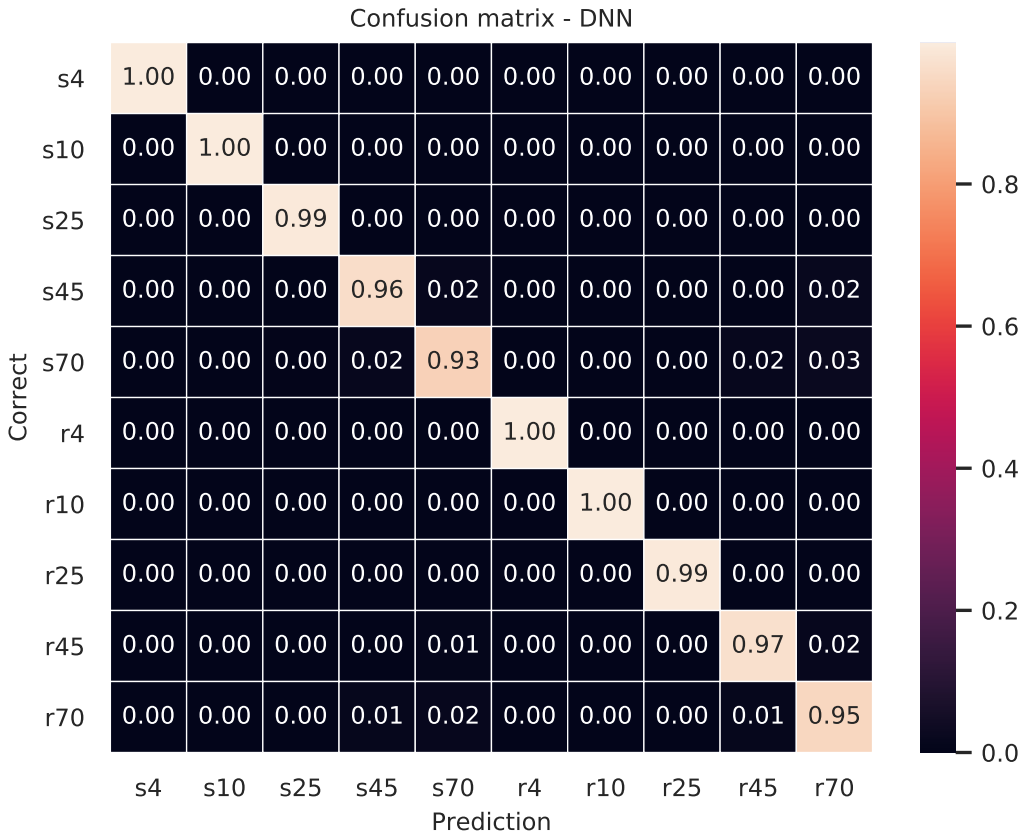


Figure 6.11: Confusion matrix for the DNN on the sawtooth LFP test set. The network struggles slightly with the higher frequency signals. It seems to struggle about equally in differentiating shape as frequency.

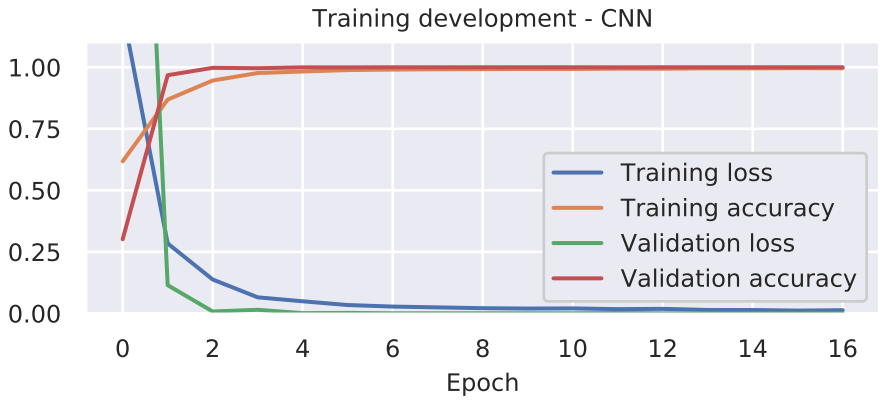


Figure 6.12: Loss/accuracy development during training for the CNN on the sawtooth LFP dataset. The reason the network seem to do better on the validation data is that training is done with dropout, while the validation is not.

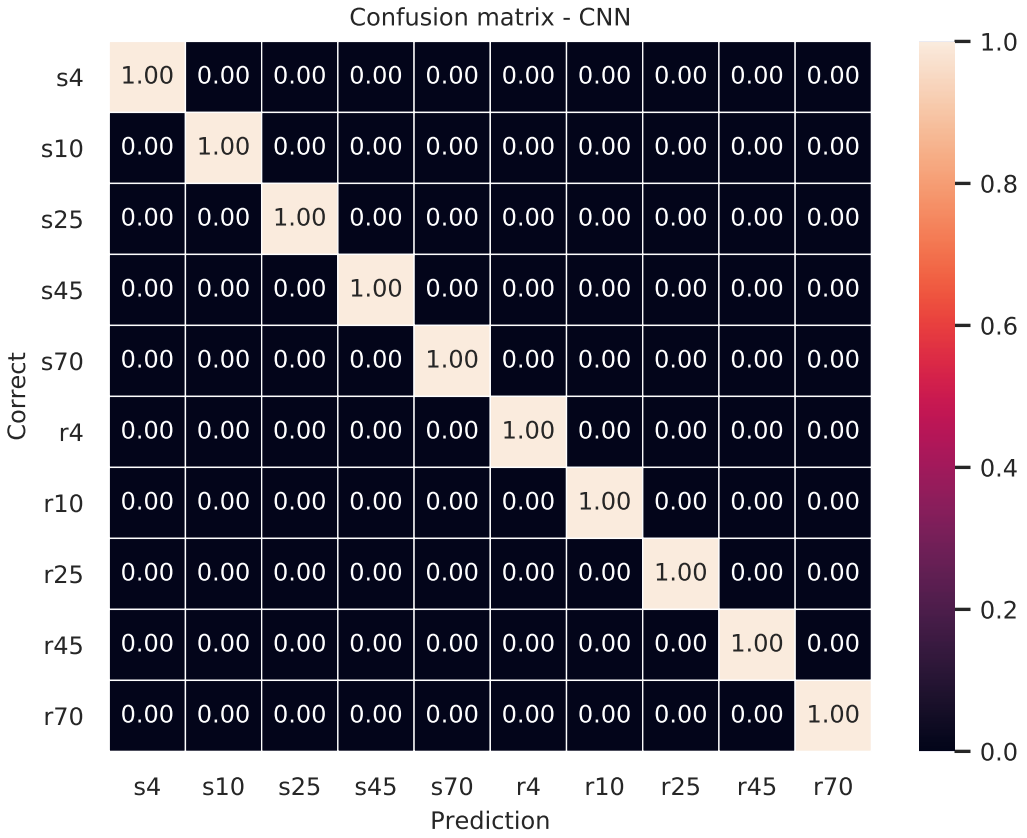


Figure 6.13: Confusion matrix for the CNN on the sawtooth LFP test set. Every single test data was correctly classified.

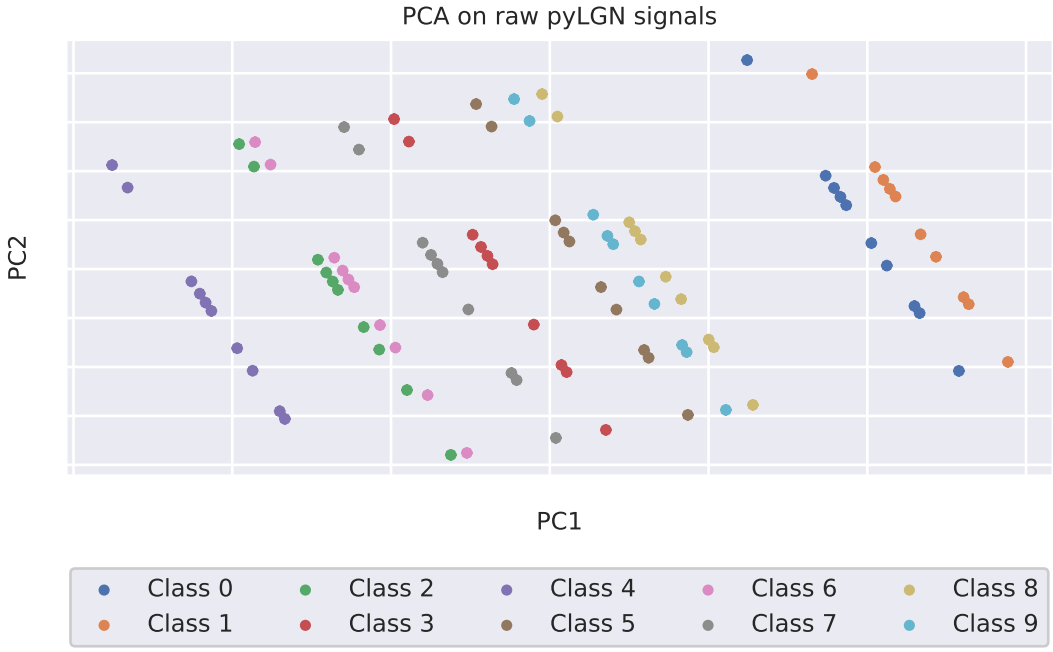


Figure 6.14: PCA on 100,000 raw pyLGN signals. The absolute most significant cause of variance in the pyLGN signal for a given image, is the image shown before. But by zooming in, we see that each point is actually a dense cluster of points, separated by tiny variances caused by the images shown as the 2nd previous, and so on. These 2nd and higher order variances are so tiny as to be negligible, meaning that there are in practice only 10 different signals per image.

6.3 Classifying raw pyLGN signals

Before we go on to classify pyLGN stimulated LFPs, we must see how well we can classify the raw output from the module. The pyLGN signal for a specific image is deterministic, but it is dependent on the state of the sheets of neurons in the module at the initial moment that it receives the image. In other words, the signal for a specific image is dependent on the previous image that was shown. Since response to the previously shown image is dependent on the image before that again, and so on, this makes it in theory a number of $\sum_{i=1}^{9} i! \approx 400\text{k}$ different signals per image.

For this reason we investigated how much error, if any, we could expect from just the pyLGN mappings of the images. We ran 10,000 pyLGN simulations, each with a randomized sequence of the images, and then cut out the part of the signal that corresponded to each image. The pieces were then ordered into 10 classes, just as we will do later with the LFPs. This made up a dataset of $10{,}000 \times 10 = 100{,}000$ signals which we then ran a PCA on. The set of images that we use as

fodder are shown in figure 6.15.

Result: None of the eventual classification error should stem from the `pyLGN` part. It turns out that in practice the 2nd and higher order effects of the image sequence are negligible. As seen in figure 6.14, the pyLGN signals data can be perfectly separated using only the two most dominant axes.

In figure 8.1 in the appendix, we have plotted 500 randomly chosen `pyLGN` signal samples for each class. The curves look pixelated because of very tiny variances, but in practice we see that there are only 10 different signals per image.

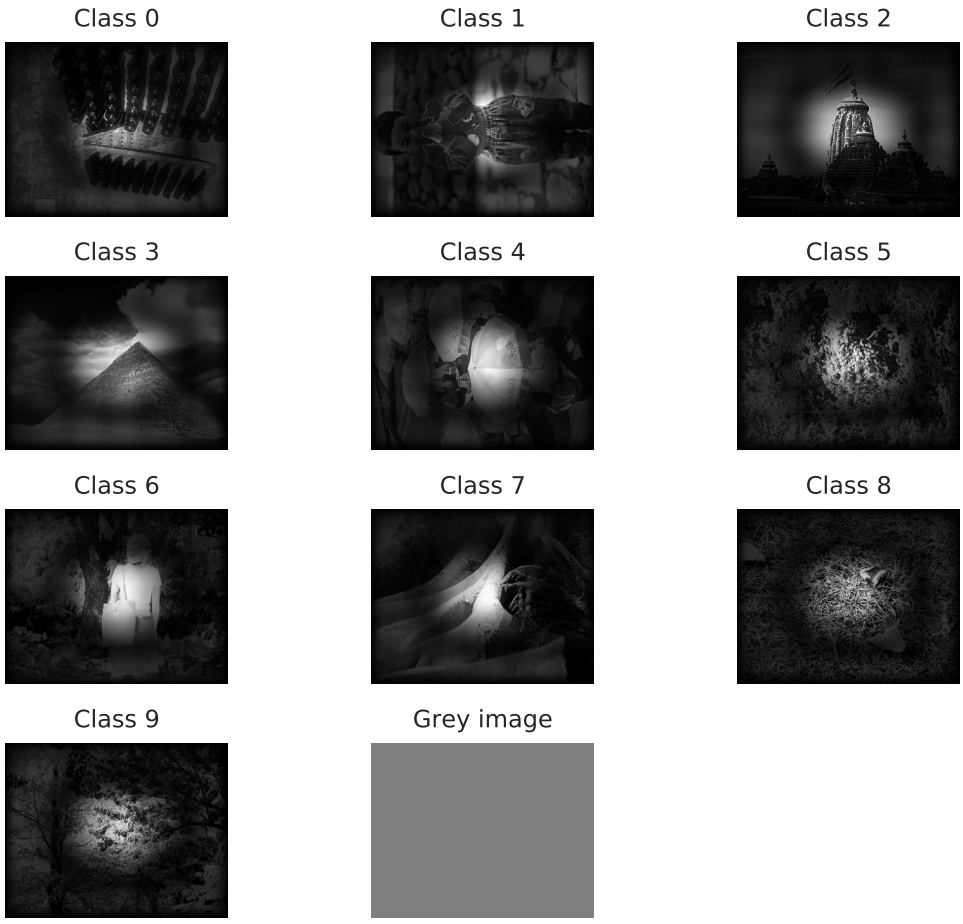


Figure 6.15: The images that were fed into PyLGN as optical stimuli for the simulation with the receptive field (RF) of the center neuron superimposed. The RF was calculated empirically by convoluting a 50×50 px black square over an all white image, and calculating the change (in L2 distance) of its `pyLGN` output signal.

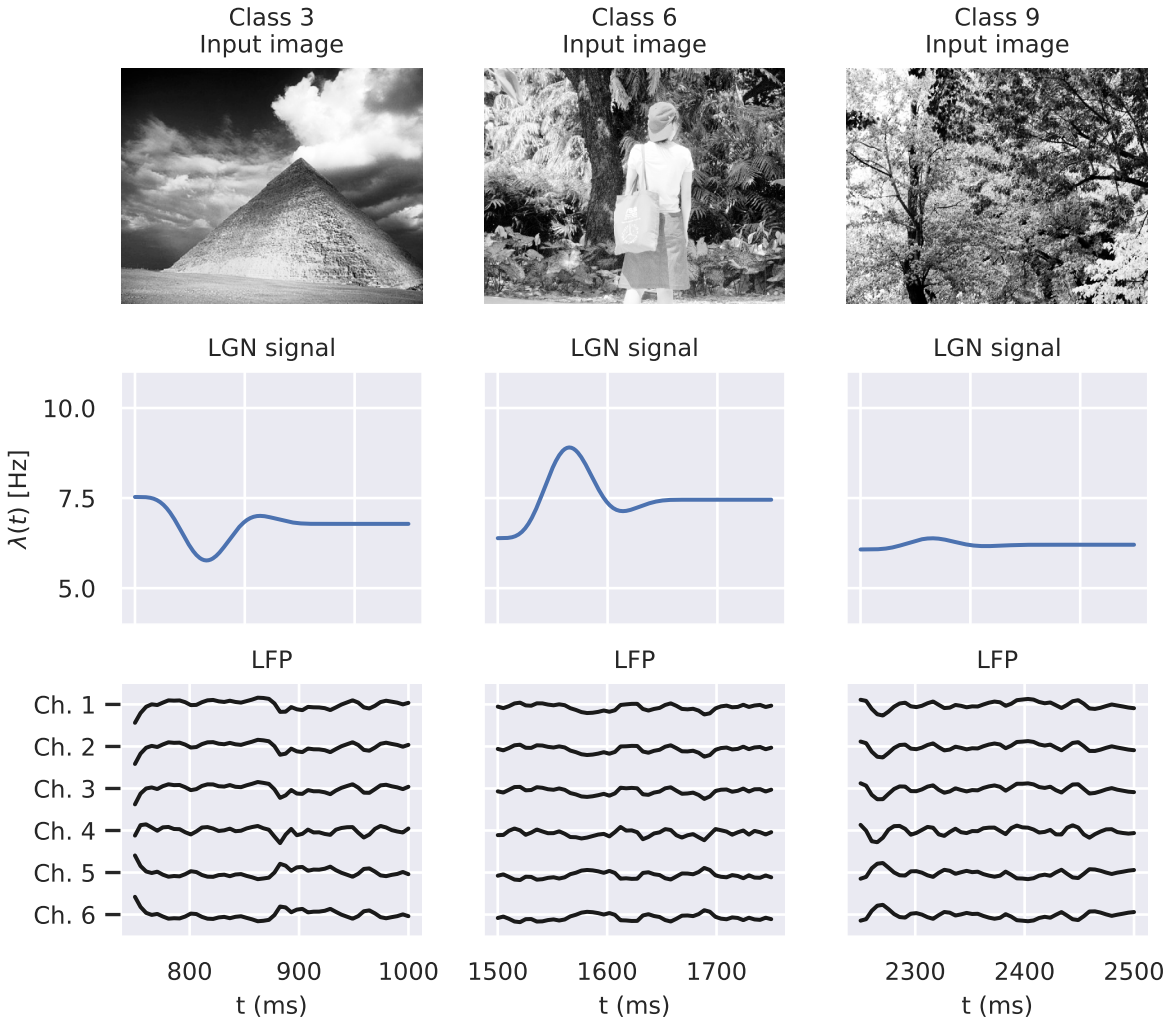


Figure 6.16: The LGN rate profiles for three of the input images and the resulting LFPs. The examples are from a simulation where all 10 images were shown for 250 ms each in the order 0, 1, 2, 3, 4, 5, 6, 7, 8, 9. The LFPs shown are after downsampling.

6.4 The full simulation

Now that we have shown that it is absolutely possible to classify the different LFPs, as well as the raw signals from `pyLGN`, we move on to the full simulation as illustrated in figure 4.1 and 1.1. We will now be using equation 4.12 as the LGN rate profile as intended, i.e.

$$\lambda_{\text{LGN}}(t) = R_{\text{LGN}}(\mathbf{0}, t). \tag{6.3}$$

We use the same set of input images as in the previous section, shown in figure 6.15. Figure 6.16 shows three examples of LGN signal and resulting LFP for three different images.

For each simulation, all 10 images are input for a duration of 250 ms each in a randomized order. For the first and last 250 ms of the simulation, the grey image is shown so that the network will stabilize before receiving the input, making a total simulation time $T = 12 \times 250 = 3000$ ms. The resolution parameters used in `pyLGN` were $nt = 12,\ nr = 5,\ dt = 1,\ dr = 0.1$, and the default DOG kernel parameter values were used.

We wanted the value of η_{tot} to have the same average as in the test phases in section 6.1 and 6.2, and oscillating with roughly same variance (which for a sinusoid is half the amplitude). By first shifting the `pyLGN` signals with the imperical mean, calculated to be about 6.5 Hz (see section 4.3.3), scaling it to have a variance of 1, and then rescaling it by $\sqrt{A_\lambda/2}$, we ensured that the `pyLGN` signals on average have a variance of $A_\lambda/2$. Again we set $A_\lambda = 3$ Hz.

The background rate was set according to equation 4.38. To avoid any confusion: this was done once, meaning that ν_{BG} was equal for all simulations.

We then ran 100k simulations. After this, the LFPs were cut, preprocessed and sorted into classes as described in section 5.1, making a total dataset of 1M datapoints of size 6×50, divided into a training, validation and test set.

6.4.1 Classification with CNN

As the CNN worked best on the sawtooth dataset, we continue with that in this section.

Result: The best of the CNN architectures we tried, managed to reach an accuracy of 66% on the test set, averaged across all classes. Although not as high as for the sawtooth data, this is quite significant as there are 10 different inputs, meaning that random guessing would give 10% accuracy. Plots with the training development and confusion matrix are shown in figure 6.17 and 6.18.

The architecture and training scheme that was used is given in table 6.3.

Again we did not perform any systematic search in parameter space, but tested the different architectures by hand. We found that there was nothing to gain by making the networks deeper and simply train them for longer, as one might expect. Smaller networks worked just as well and were of course much faster.

In addition, increasing the receptive field of the nodes in the convolutional layers (see section 3.4.1) had, surprisingly, no effect on the accuracy. This might indicate that most of the learning happens in the dense layers of the network. We confirmed this by training an additional DNN on the data, which came up close at a 60% average accuracy (we do not see it as important to report on the architecture for this). However, adding additional dense layers to the CNN gave no additional gain in accuracy.

The accuracy for the individual images varied quite a bit, as shown in the confusion matrix in figure 6.18. We can see that there are some images that the CNN struggles to separate from each other more than others. For example, whenever the CNN sees an LFP generated from image 9, it is almost as likely to say that it is image 5 or 8. We discuss this further in the last chapter.

No issues were had with overfitting, so the regularization was kept low and dropout was not used.

Classification scheme	Value
Algorithm	Convolutional Neural Net (CNN)
# conv. layers	2
Kernel sizes	11, 11
# kernels per layers	32, 32
Strides	1, 1
Maxpool	Yes
- Pool size	2
Dropout	None
# dense layers	1
# nodes per dense layer	128
Activation	ReLu
Output activation	Softmax
Optimizer	Adam
- Learning rate	0.001
- Decay	0.002
L2 regularization	0.01
Batch size	500
Earlystopping patience	5

Table 6.3: Network and training scheme for the best CNN on the LFP data from the full simulation.

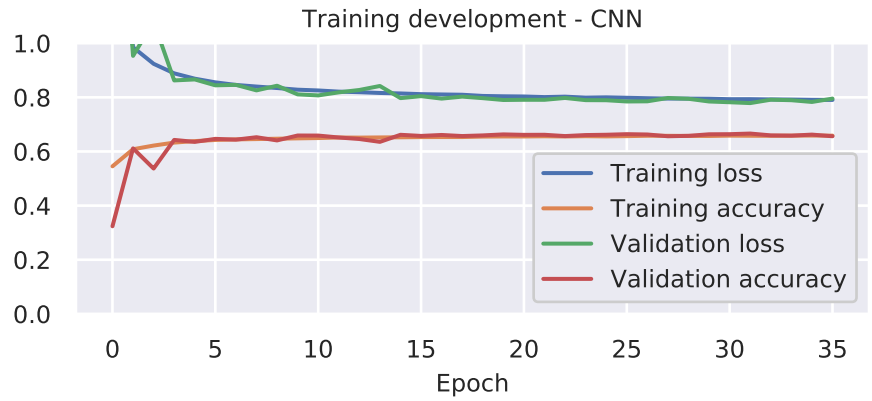
6.4.2 Classification with LSTM

After seeing the results of the CNN, we decided to try an LSTM as well. The two algorithms are very different in the way they perceive the input, which can potentially make them differently suited for various sorts of data. The LSTM is slower to train and use than the CNN, but it is interesting to see if there are any different results in performance of the two algorithms.

Result: The performance of the LSTM was virtually the exact same as the CNN, reaching a 65% average accuracy on the test set. The confusion matrix (figure 6.20) is also nearly identical to that of the CNN, meaning that exact same images were confused to the same degrees. The main difference was that the LSTM was over a magnitude slower to train. Here too we saw that using a larger network had no positive effect on the accuracy.

Classification scheme	Value
Algorithm	Long Short-Term Memory (LSTM)
# LSTM layers	1
- Hidden layer size	128
- Activation	tanh
- Recurrent activation	Hard sigmoid
Output activation	Softmax
Dropout	None
Optimizer	Adam
- Learning rate	0.001
- Decay	0.003
L2 regularization	0.01
Batch size	500
Earlystopping patience	5

Table 6.4: Network and training scheme for the best LSTM on the LFP data from the full simulation.



[h]

Figure 6.17: Training development of the CNN on the `pyLGN` stimulated LFP dataset. The reason it looks like it converges faster than in the test sections, is because the dataset is 10x bigger here, meaning more training is done per epoch.

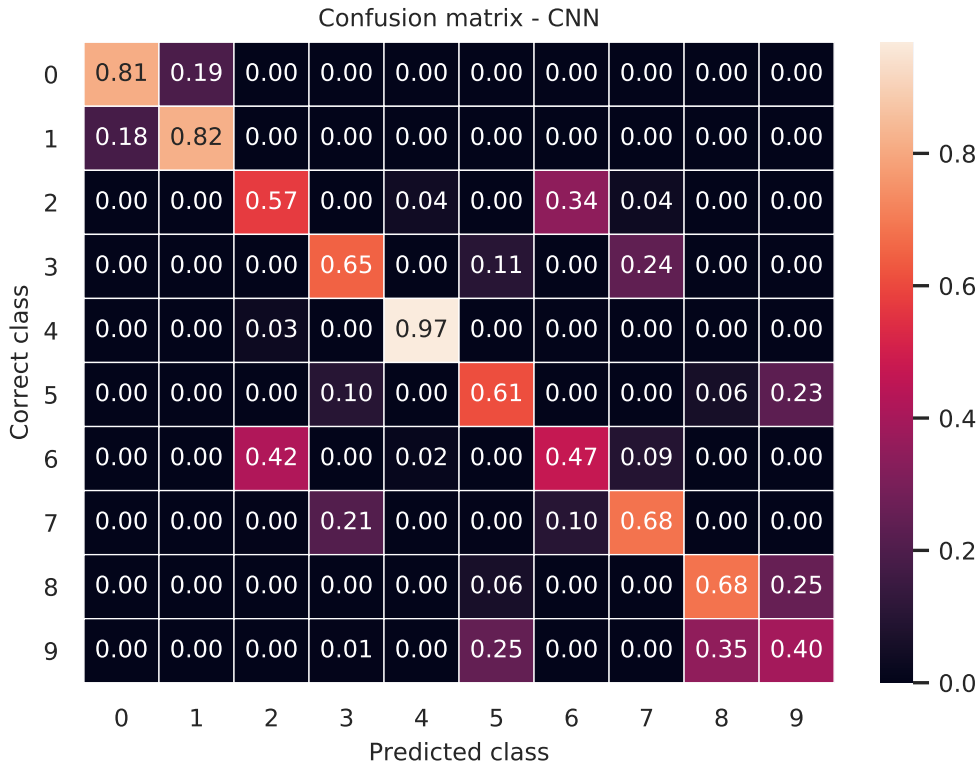
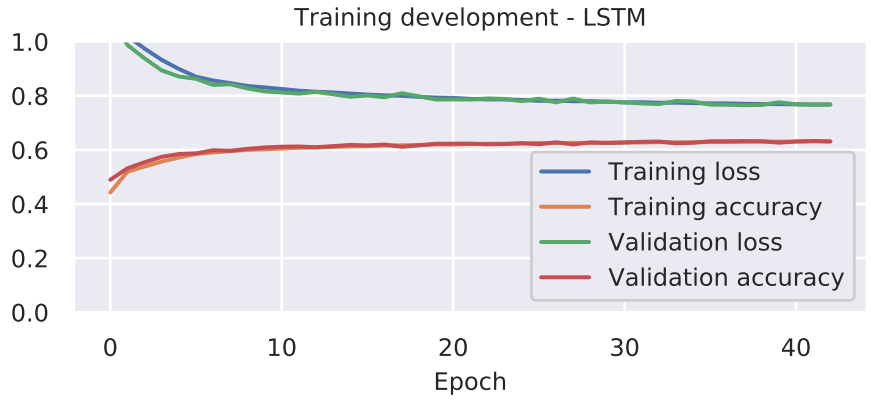


Figure 6.18: Confusion matrix of the CNN on the `pyLGN` stimulated LFP dataset.



[h]

Figure 6.19: Training development of the LSTM on the pyLGN stimulated LFP dataset.

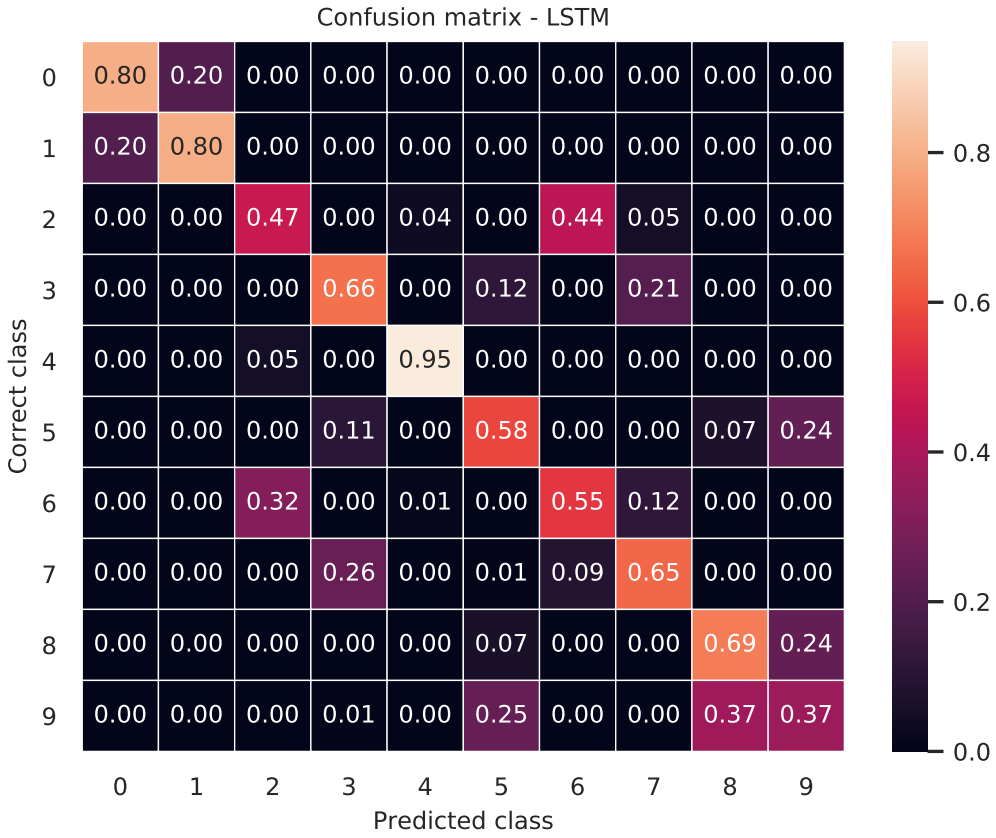


Figure 6.20: Confusion matrix of the LSTM on the pyLGN stimulated LFP dataset.

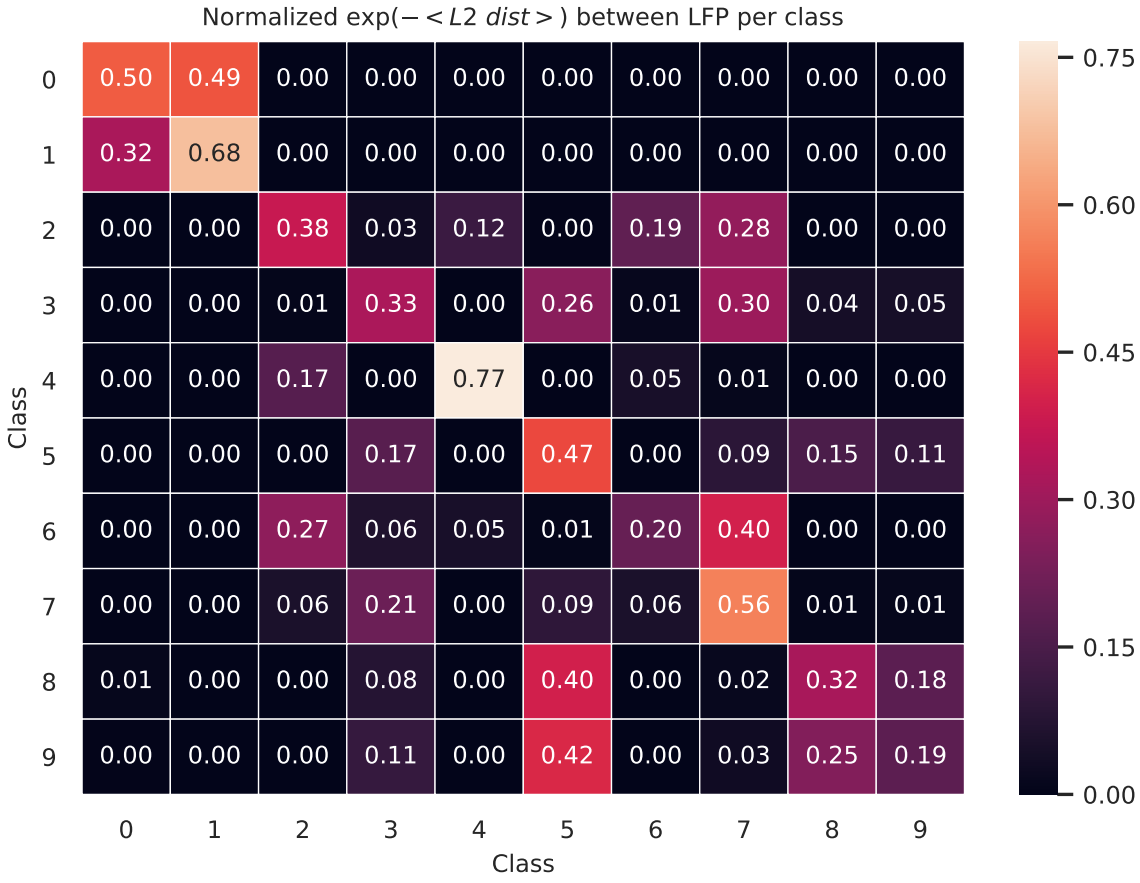


Figure 6.21: The mean L2 distance between the classes, mapped through an exponential function to correspond to values between 0 and 1. Higher values means lower L2 distance.

6.4.3 Investigating what the AIs look for in the LFP

A much discussed issue with deep learning is that it is often hard to know what exactly it is that the classifiers are looking for in the input. Scientists have for example been surprised to find that AIs trained to recognize objects in images are surprisingly concerned with texture, compared to shapes [14]. Knowing what features it is that AIs use to make their decisions can be crucial for improving the algorithms further, or for avoiding exploitation [38], and is in general an obviously important field of study.

As an approach to investigate what the ANNs look for in the LFP data, we experimented with various measures of distance between the different classes. We found that the L2 distances between the LFPs somewhat resembled the pattern of the confusion matrices in figure 6.18 and 6.20.

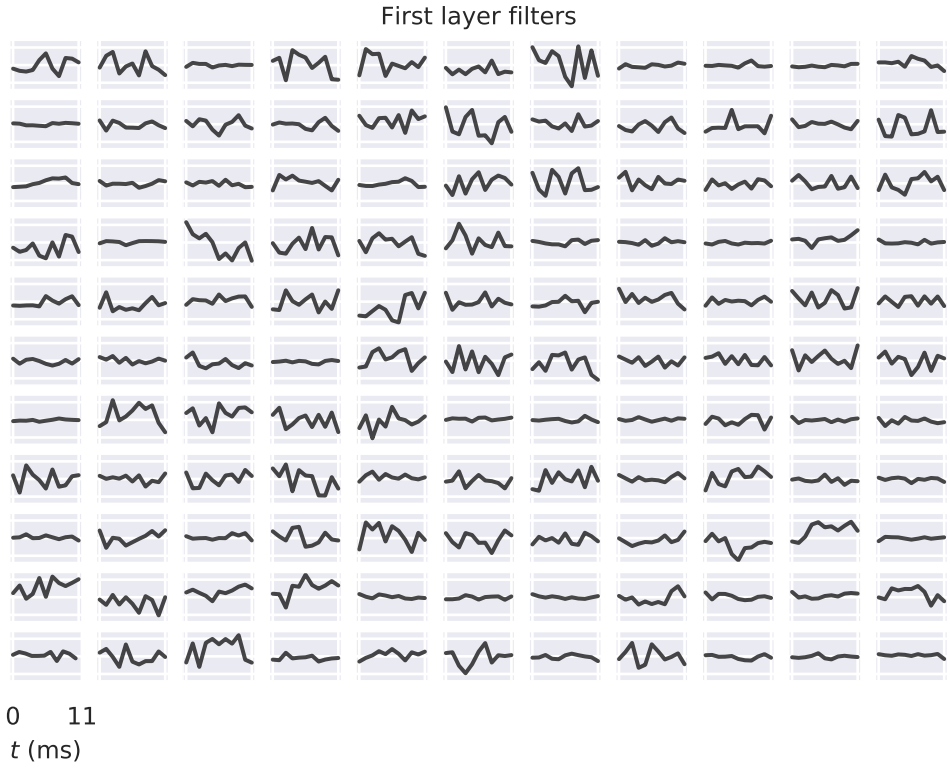


Figure 6.22: 121 of the kernels/filters of the first layer in the trained CNN.

This was done in the following way: For a given class i, the mean L2 distance to class j was calculated and mapped to a value between 0 and 1 through the equation

$$\langle L2 \rangle_{ij} = \exp\Big[-\frac{1}{N^2} \sum_{k=1}^{N} \sum_{l=1}^{N} \big(\boldsymbol{x}^{ki} - \boldsymbol{x}^{lj} \big)^2 \Big]. \tag{6.4}$$

The vector $\boldsymbol{x}^{ki}$ contains the k'th LFP data of class i (for simplicity, only data from channel 1 was used). This calculation was then done for all combinations of ij and put into a matrix. In order to compare this to the confusion matrices, each row in the matrix was then normalized to sum to 1. The matrix with the mapped L2 distances is plotted in figure 6.21, where a high number represents a small average distance. Because of the exponential mapping, the plot only indicates differences in the very small distances. All longer distances are simply clamped to 0.

For the CNN, another common thing to do in order to investigate what it

looks for, is to inspect the filters of the first convolutional layer. The shape of the filter is the shape of the LFP that would maximize the output of that particular layer, which means that this is the pattern that the layer "looks for" in the LFP.

In figure 6.22, 121 of the first layer filters of the trained CNN are visualized. As expected, there is a manifold of patterns that the network look for, and many seem rather flat, which could potentially mean that they are redundant and that we could get away with fewer kernels. Unfortunately we cannot do this analysis with the LSTM.

6.5 Additional Experiments

6.5.1 Fixed Connectome

In the point network simulations that were used in all previous sections, the connections in the network – the connectome – were randomized each time. We generated a new dataset where we used an identical connectome for all the simulations and trained new ANNs on this.

Result: The results were indistinguishable. This should be expected, especially considering that we generate the LFP based on the population firing rates. As the network gets larger this should be less and less dependent on the specific connectome.

6.5.2 Extra noisy test set

To compare the robustness of the now trained CNN and LSTM against noise, we tested their accuracies on the test set while we gradually added more and more Gaussian noise to it. Starting at 0.01, the std was increased step-by-step up to 0.5, where the data became completely unclassifiable. (The noise was added to the fully preprocessed and downsampled data.)

Result: As seen in figure 6.23, the LSTM seem to be slightly better at handling the noise than the CNN.

6.5.3 Double LGN amplitude

We generated a full dataset where the strength of the LGN signal was twice that of in the previous section, i.e. $A_\lambda = 6$ Hz. The value of η_{tot} was held at 1.1 by setting the background rate according to equation 4.38. We trained new classifiers on this and tested them on the new test set.

Result: As expected this gave a higher average classification accuracy, which was now at 70%.

6.5.4 Test Set with Other g Values

If the goal is to train ANNs on simulated data in order to classify experimental data, it is interesting to ask ourselves what happens if we are mistaken about one or more of the parameter values in the biological brain that we simulate. The question is then, how does the classifiers transfer to data that from a brain with slight deviations in parameters?

To begin to investigate this, we generated two new test sets where we changed the parameter g by $\pm 10\%$. We then let the ANNs attempt to classify these sets. In other words, we took the CNN and LSTM that were trained on the original data, and ran predictions on test sets from simulations where $g = 4.68$ and $g = 5.72$.

The new test datasets were preprocessed using the mean and stds from the original training set, which meant that they did not have a mean of zero or unitary standard deviation, as seen in figure 6.24.

Result: The CNN transfered much better to the new data than the LSTM for both new values of g, as seen in figure 6.25 and 6.26. In fact, on average the CNN did almost as well when g was 10% higher as for the original value, with a 62% accuracy. Both classifiers also transfered better when g was raised than lowered.

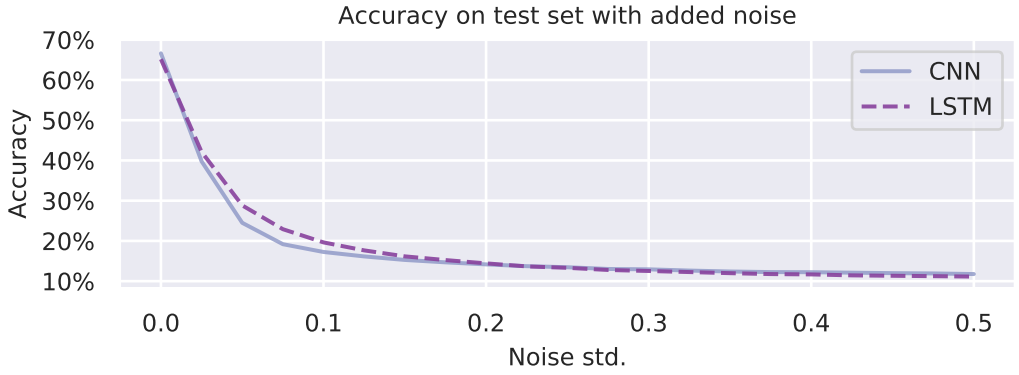


Figure 6.23: The accuracy of the CNN and LSTM as we gradually increased the standard deviation of the added noise.

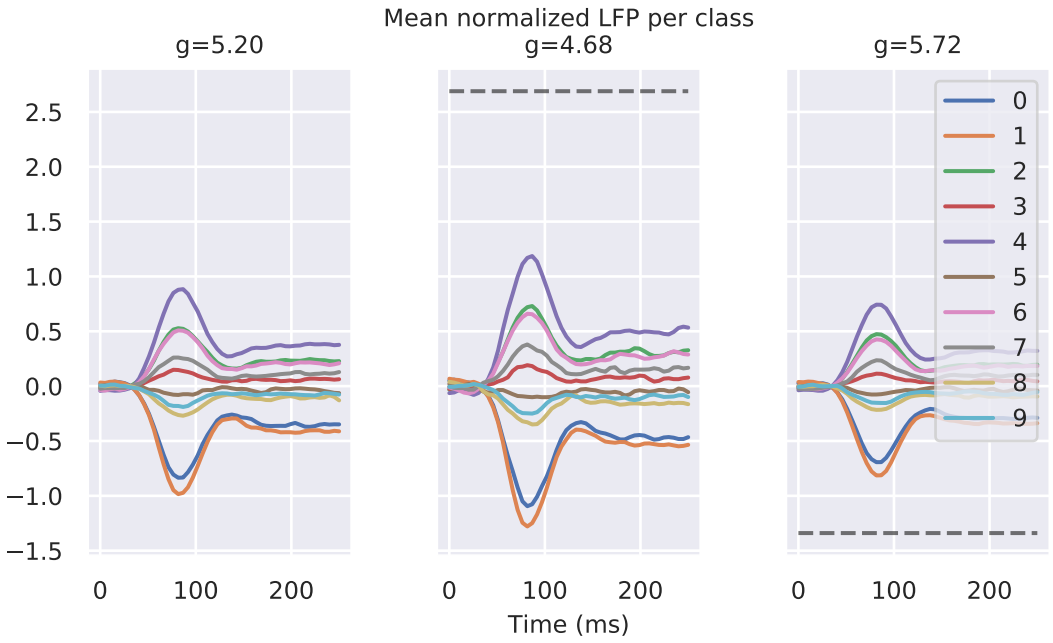


Figure 6.24: Mean LFP per class for channel 1 for the three different values of g. As all the test set were shifted and scaled using the mean and variance from the original training set, the two new test sets were not initially zero centered. However, for this specific plot they were zero centered to show how the shape is changed with new g values, and the dashed black lines show the mean the signals were zentered around at test time.

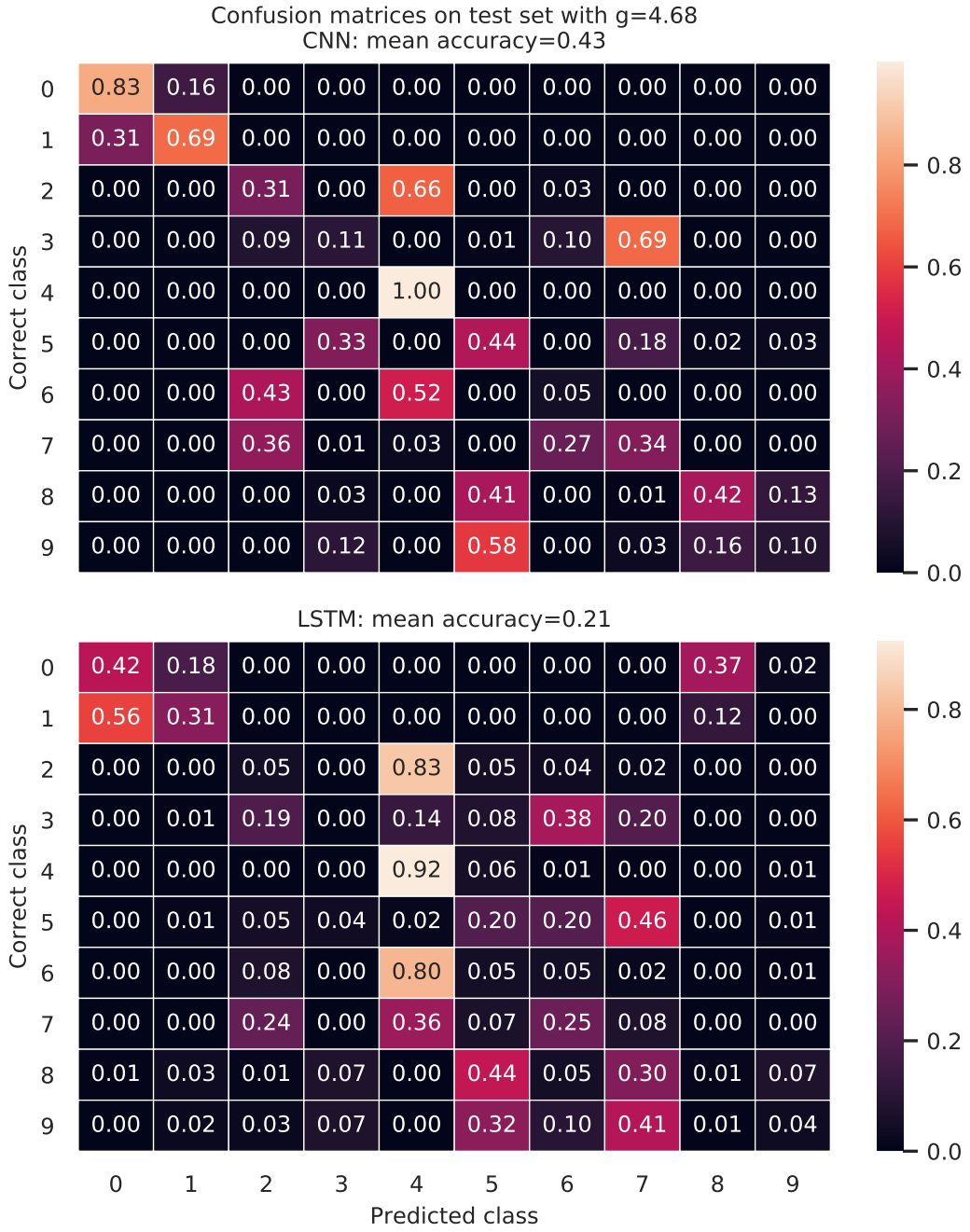


Figure 6.25: Confusion matrices for the CNN and LSTM on the test dataset where the value of g was lowered by 10%.

Figure 6.26: Confusion matrices for the CNN and LSTM on the test dataset where the value of g was raised by 10%.

Chapter 7

Conclusion

7.1 Discussion

The results of the experiments that we have performed here suggest that we can indeed use the LFP to extract knowledge about the information that a brain is processing. How much of this information we can get also seems, unsurprisingly, dependent on the strength of the communication between the relevant neuron populations, here referring to the size of the LGN firing rate compared to the background rate. We could likely get an much higher classification accuracy by simply turning up the amplitude of the LGN signal, but this would in itself not be very interesting. Systematically analyzing the relationship between the amplitude and the accuracy could on the other hand be something worthy of investigation.

CNN vs LSTM

We have compared the performance of two very different artificial neural network algorithms, namely a CNN and an LSTM. The two are different in the way they perceive the input. The mechanisms of a CNN is analogous to vision, where everything in the input is perceived at the same time in parallel. On the other hand, the way we use the LSTM here is somewhat more analogous to hearing, where the values in the input data are read one at a time in series – a little like how an ear reads sound from a train of different air pressure values. This could make us expect that the two algorithms would pick up different sorts of patterns and therefore respond differently to various features in the data. One might for example expect the LSTM to be better at picking up causal effects, but their performances were strikingly similar.

An important difference was that the CNN transfered better to data from different networks, i.e. simulations with different g values. The LSTM seems to

become much more biased to a small subset of the classes, compared to the CNN. We see for example that in figure 6.25, the LSTM becomes very keen on placing data into class 4, while in figure 6.26 it favours class 1 and 8. It is not obvious why this is the case, but one possible explanation could be that the CNN is more affected by the mean values of the input, which by looking at figure 6.24 we see are scaled when g is changed. This explanation would also fit with the result that the LSTM handles Gaussian noise better, as the mean is then held constant.

With both networks, increasing their sizes by adding more layers and nodes gave very diminishing returns. A very small CNN could get up to 65% average accuracy, and adding layers, magnifying the convolutional filters etc. would improve it by 1% but no more.

The LSTM was, in addition to being slower to train and worse at transferring, more dependent on finding a good tuning of its hyperparameters. By that we mean that whereas many different CNN architectures that we tested would perform about equally well, the LSTM seemed to have fewer hyperparameter configurations that would give a good result. As already said, the LSTM weighed up for some of this by being better at noisy data.

Transfer learning

It is not obvious why raising g by 10% would make the data easier to classify, compared to lowering it by the same amount. Looking at figure 2A in [1], it could possibly have something to do with the fact that the border between the AI and SI domain, falls downwards in the area of $4 \geq g \geq 5$. As our network is oscillating up and down slightly above this border, it is possible that for the higher value of g, it would occasionally cross over into the SI domain. If data from this state is easier to classify, this could be a possible explanation. We did not test this, however.

Visual similarities in the confused classes

By comparing figure 6.18 and 6.15, we can see that the classes that the network confuse are images that in many ways look visually similar. Taking again the examples of image 5, 8 and 9, we see that these three images all have very similar textures inside the field of view. Additionally, none of them have any larger objects to speak of.

Two other classes that the ANNs confuse are image 6 and 2. We see that both of these have a large vertically elongated object placed right in the receptive field.

Another pair that creates some confusion consists of image 3 and 7, on which

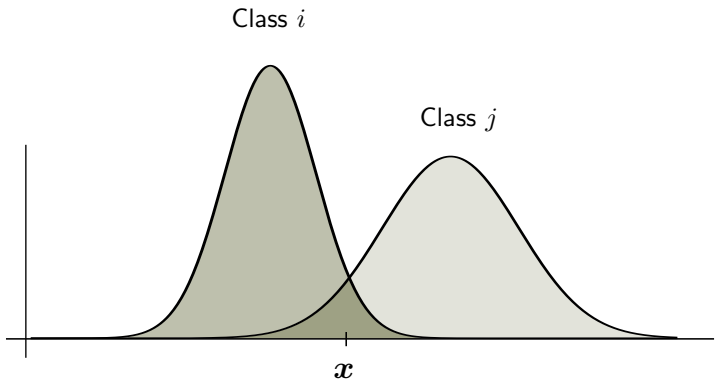


Figure 7.1: Illustrating two overlapping distributions. Any datapoint $\boldsymbol{x}$ from the overlapping area could stem from both classes and would not be classifiable.

we can see somewhat similar triangular shapes.

The odd pair out is probably class 0 and 1, which the networks confuse to some degree but which at first glance seem less visually similar than the sets mentioned above. This does therefore remind us to not to be too confident in pointing out the visual similarities, as it is easy to be prone to confirmation bias here. At the same time, we do see that they both are somewhat contain some larger horizontal shape that looks nothing like the other images, and this could perhaps be what causes the networks to confuse them.

In terms of accuracy, class 4 clearly stands out at 97 and 95%. Looking at the images again, we can see that it is also probably the one that sticks out the most visually. It has a multiple of of semi-large, oval objects which look nothing like anything in the other images, and it does not have any of the fine grained texture that most of the others have.

The rate at which the classifiers confuse the classes seems to be partly explained by the L2 distance between them, as indicated by similarities of the pattern in figure 6.21 and the confusion matrices in figure 6.18 and 6.20. We see, for example, that LFPs from class 6 and 2, which are the two classes the ANNs confuse the most, have on average a relatively small L2 distance between them. At the same time, so does 7 and 2, and the network does not have much problems separating those. We therefore do not make any clear conclusions on this.

Overlapping LFP distributions

Both classifiers were about equally successful at 65 and 66% on the initial dataset. They could also reach this accuracy in very few epochs and with many different hyperparameter settings. This maye therefore indicate that the LFP data is an easy classification task for ANNs, but that the distribution of LFPs for the

various classes are overlapping.

By that we mean that it could be the case that some of the data from class i could be indistinguishable, even in principle, from certain datapoints from class j. In other words, for a given LFP signal created when image i was shown, it could be that, because of the large degree of stochasticity in the simulation, that the very same signal could also be created when image j was shown, making those specific datapoints literally unclassifiable. The concept is illustrated in figure 7.1 above.

It is plausible that this is the reason that the ANNs were not able to correctly classify much of the data. It would otherwise seem unlikely that 65% of the data should be extremely easy, while the remaining 35% extremely hard.

7.2 Future prospects

A natural extension of this project would be to run simulations using bigger networks and more realistic parameter configurations. Multiple LGN rate profiles could for example be used, with a manifold of LGN sub-pupulations to stimulate cortex.

By tuning the ratio of LGN signals to background noise to conform with some equivalent measure from experiments, the realism of the simulation could be enhanced even further. The ANNs trained on data from this would be much more likely to transfer well to experimental LFP data.

Additional work could be done to make the trained ANNs more robust, for example by generating training data where the simulation parameters are drawn from distributions, rather than being fixed.

Chapter 8

Appendix

8.1 PCA

The core of Principal Component Analysis (PCA) is about finding the axes in our dataspace that explains the most of the data's variance. This can be very useful for reducing the dimensionality of the data. The proof of the algorithm goes as the following:

We have our data stored in a matrix $\boldsymbol{X} \in \mathbb{R}^{DxN}$, consisting of N datapoints that are columns vectors of length D. We assume that the data is **zero-centered**.

Let us now assume that there exists an orthonormal rotation matrix $\boldsymbol{P} = [\boldsymbol{p}_1\ \boldsymbol{p}_2 \dots\ \boldsymbol{p}_D]$ that can transform our data matrix into $\boldsymbol{Y} \in \mathbb{R}^{DxN}$ where $\boldsymbol{Y}$ has the property that its covariance matrix is diagonal. In other words,

$$\boldsymbol{Y} = \boldsymbol{P}^T \boldsymbol{X} \tag{8.1}$$

where

$$\mathrm{Cov}(\boldsymbol{Y}) = \begin{pmatrix} \lambda_1 & & 0 \\ & \ddots & \\ 0 & & \lambda_D \end{pmatrix}. \tag{8.2}$$

Since the data is zero-centered, we have $\mathrm{Cov}(\boldsymbol{X}) = \frac{1}{N}\boldsymbol{X}\boldsymbol{X}^T$ and similarly $\mathrm{Cov}(\boldsymbol{Y}) = \frac{1}{N}\boldsymbol{Y}\boldsymbol{Y}^T$. Inserting equation 8.1 into the latter, we get

$$\begin{aligned} \mathrm{Cov}(\boldsymbol{Y}) &= \frac{1}{N}(\boldsymbol{P}^T\boldsymbol{X})(\boldsymbol{P}^T\boldsymbol{X})^T \\ &= \frac{1}{N}\boldsymbol{P}^T\boldsymbol{X}\boldsymbol{X}^T\boldsymbol{P} \\ &= \boldsymbol{P}^T\,\mathrm{Cov}(\boldsymbol{X})\boldsymbol{P}. \end{aligned} \tag{8.3}$$

Multiplying both sides of equation 8.3 with $\boldsymbol{P}$ and using the fact that $\boldsymbol{P}\boldsymbol{P}^T = \boldsymbol{I}$ because of its orthonormality, we get

$$\boldsymbol{P}\,\mathrm{Cov}(\boldsymbol{Y}) = \mathrm{Cov}(\boldsymbol{X})\boldsymbol{P}. \tag{8.4}$$

Substituting the left $\boldsymbol{P}$ with $[\boldsymbol{p}_1\ \boldsymbol{p}_2 \ldots\ \boldsymbol{p}_D]$ and $\mathrm{Cov}(\boldsymbol{Y})$ with the right side of expression 8.2, we see that can rewrite the above equation as

$$[\lambda_1\boldsymbol{p}_1\ \lambda_2\boldsymbol{p}_2\ \ldots\ \lambda_D\boldsymbol{p}_D] = \mathrm{Cov}(\boldsymbol{X})\boldsymbol{P}. \tag{8.5}$$

We then for simplicity rename $\mathrm{Cov}(\boldsymbol{X}) = \boldsymbol{Z} = [\boldsymbol{z}_1 \ldots \boldsymbol{z}_D]$ so we can write

$$\begin{aligned}[\lambda_1\boldsymbol{p}_1\ \lambda_2\boldsymbol{p}_2\ \ldots\ \lambda_D\boldsymbol{p}_D] &= \boldsymbol{Z}\boldsymbol{P} \\ &= [\boldsymbol{Z}\boldsymbol{p}_1\ \boldsymbol{Z}\boldsymbol{p}_2\ \ldots\ \boldsymbol{Z}\boldsymbol{p}_D]\end{aligned} \tag{8.6}$$

From this we see that we get the relation

$$\lambda_i\boldsymbol{p}_i = \boldsymbol{Z}\boldsymbol{p}_i \ \text{ for each } \boldsymbol{p}_i. \tag{8.7}$$

In other words, each $\boldsymbol{p}_i$ is an eigenvector of $\boldsymbol{Z}$ with corresponding eigenvalue λ_i.

To sum up: The rotation matrix $\boldsymbol{P}$ that rotates $\boldsymbol{X}$ into a matrix $\boldsymbol{Y}$ which has a diagonal covariance matrix, can be found by simply calculating the eigenvectors of $\boldsymbol{Z} = \mathrm{Cov}(\boldsymbol{X})$.

The vector $\boldsymbol{p}_i$ is the i'th **principal component** of our data. The corresponding eigenvalue λ_i is the variance in dimension i in $\mathrm{Cov}(\boldsymbol{Y})$, meaning is the amount of variance along axis $\boldsymbol{p}_i$. This means that if λ_a is the largest eigenvalue, then $\boldsymbol{p}_a$ is the direction in data space which explains the most of the variance. If λ_b is the second largest eigenvalue, then $\boldsymbol{p}_b$ is the direction in data space which explains the second most of the variance, and so forth.

Most importantly, since the $\boldsymbol{P}$ is an orthonormal matrix, the principal components are orthogonal and are thus uncorrelated. That means there is no overlap between the variance that is explained by two different principal components.

You can now reduce the D dimensional data down to K dimensions by checking which are the K largest eigenvalues, and then put their corresponding eigenvectors into a new matrix $\boldsymbol{P}_{\text{reduced}} = [\boldsymbol{p}_a\ \boldsymbol{p}_b\ \ldots]$. This can now be used to project the data matrix $\boldsymbol{X}$ onto the K biggest principal components by

$$\boldsymbol{Y}_{\text{reduced}} = \boldsymbol{P}^T_{\text{reduced}}\boldsymbol{X}. \tag{8.8}$$

$\boldsymbol{Y}_{\text{reduced}}$ will now contain a lower dimensional representation of the data.

8.2 pyLGN Signals

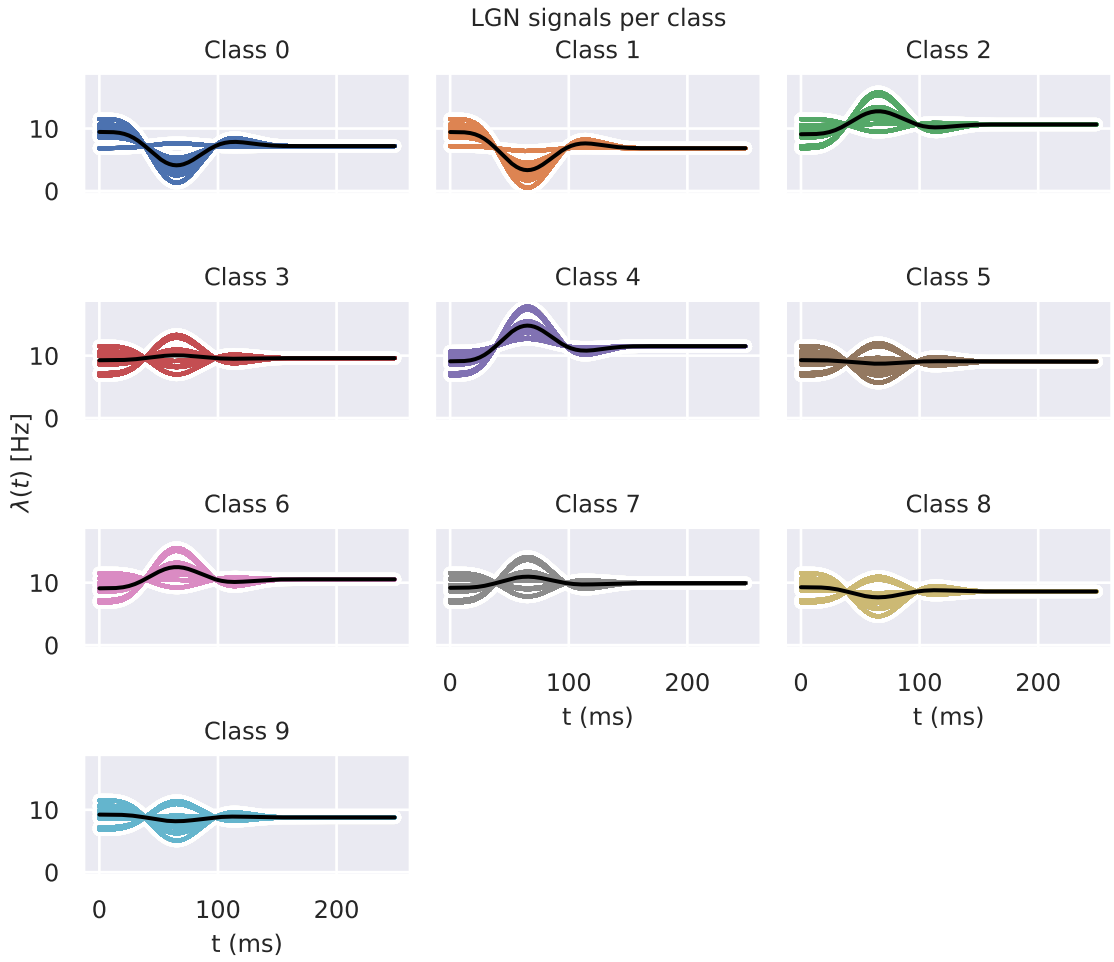


Figure 8.1: 500 examples of `pyLGN` signals for each class. Each signal is taken from a `pyLGN` simulation using a randomized sequence of the 10 images. For each simulation the ouput signal was then into pieces of 250 ms corresponding to each image. The only significant variation in the signals stem from which image was shown as the previous image. There are thus in practice exactly 10 signals for each image. The black curve is the mean over the 500 signals.

References

[1] Nicolas Brunel. Dynamics of networks of randomly connected excitatory and inhibitory spiking neurons. *Journal of Physiology Paris*, 94(5-6):445–463, 2000.

[2] J Carey. Brain facts, Fourth Edition. 2003.

[3] Alexander Casti, Fernand Hayot, Youping Xiao, and Ehud Kaplan. A simple model of retina-LGN transmission. *Journal of Computational Neuroscience*, 24(2):235–252, 2008.

[4] François Chollet et al. Keras. https://keras.io, 2015.

[5] James J. Cochran, Louis A. Cox, Pinar Keskinocak, Jeffrey P. Kharoufeh, J. Cole Smith, and Raghu Pasupathy. Generating Nonhomogeneous Poisson Processes. *Wiley Encyclopedia of Operations Research and Management Science*, pages 1–11, 2011.

[6] G. T. Einevoll and H. E. Plesser. Linear mechanistic models for the dorsal lateral geniculate nucleus of cat probed using drifting-grating stimuli, 2002.

[7] Gaute T Einevoll and Henrik Lindén. Local Field Potentials: Biophysical Origin and Analysis. In *Principles of Neural Coding*, number January, chapter 3, pages 37–60. 2013.

[8] Gaute T. Einevoll and Hans E. Plesser. Extended difference-of-Gaussians model incorporating cortical feedback for relay cells in the lateral geniculate nucleus of cat. *Cognitive Neurodynamics*, 6(4):307–324, 2012.

[9] B Y Christina Enroth-cugell and J G Robson. The Contrast Sensitivity of Retinal Ganglion Cells of the Cat. *The Journal of Physiology*, 187(3):517–552, 1966.

[10] Amidi et al. RNN: Recurrent neural networks cheatsheet. https://stanford.edu/~shervine/teaching/cs-230/cheatsheet-recurrent-neural-networks. Stanford University, CS 230.

[11] Martín Abadi et al. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[12] Virtanen et al. SciPy 1.0–Fundamental Algorithms for Scientific Computing in Python. *arXiv e-prints*, page arXiv:1907.10121, Jul 2019.

[13] M Garrett. Allen Brain Observatory: Visual Coding. 0(June):1–24, 2016.

[14] Robert Geirhos, Claudio Michaelis, and Patricia Rubisch. I Mage N Et - Trained Cnn S Are Biased Towards Texture ; Increasing Shape Bias Improves. (c):1–22, 2019.

[15] Marc-Oliver Gewaltig and Markus Diesmann. Nest (neural simulation tool). *Scholarpedia*, 2(4):1430, 2007.

[16] Espen Hagen, David Dahmen, Maria L. Stavrinou, Henrik Lindén, Tom Tetzlaff, Sacha J. Van Albada, Sonja Grün, Markus Diesmann, and Gaute T. Einevoll. Hybrid scheme for modeling local field potentials from point-neuron networks. *Cerebral Cortex*, 26(12):4461–4496, 2016.

[17] Thomas Heiberg, Birgit Kriener, Tom Tetzlaff, Alex Casti, Gaute T. Einevoll, and Hans E. Plesser. Firing-rate models capture essential response dynamics of LGN relay cells. *Journal of Computational Neuroscience*, 35(3):359–375, 2013.

[18] Michael L Hines. The NEURON Simulation Environment. In *The Handbook of Brain Theory and Neural Networks*. 2002.

[19] A. L. Hodgkin and A. F. Huxley. A quantitative description of membrane current and its application to conduction and excitation in nerve. *Bulletin of Mathematical Biology*, 1990.

[20] Gary R. Holt and Christof Koch. Electrical interactions via the extracellular potential near cell bodies. *Journal of Computational Neuroscience*, 6(2):169–184, 1999.

[21] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *32nd International Conference on Machine Learning, ICML 2015*, 1:448–456, 2015.

[22] Keith B. Hengen*, Mary E. Lambo, Stephen D. Van Hooser*, Donald B Katz#, Gina G, and Turrigiano. Firing Rate Homeostasis in Visual Cortex of Freely Behaving Rodents. *Bone*, 23(1):1–7, 2008.

[23] Diederik P. Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization. pages 1–15, 2014.

[24] Michal Kosinski and Wang Wang. Deep Neural Networks Are More Accurate Than Humans at Detecting Sexual Orientation From Facial Images. *Journal of Personality and Social Psychology*, 114(2):246–257, 2018.

[25] Henrik Lindén, Espen Hagen, Szymon Łeski, Eivind S. Norheim, Klas H. Pettersen, and Gaute T. Einevoll. LFPy: A tool for biophysical simulation of extracellular potentials generated by detailed model neurons. *Frontiers in Neuroinformatics*, 7(JAN):1–15, 2014.

[26] Wenjie Luo, Yujia Li, Raquel Urtasun, and Richard Zemel. Understanding the effective receptive field in deep convolutional neural networks. *Advances in Neural Information Processing Systems*, (Nips):4905–4913, 2016.

[27] Pankaj Mehta, Marin Bukov, Ching Hao Wang, Alexandre G.R. Day, Clint Richardson, Charles K. Fisher, and David J. Schwab. A high-bias, low-variance introduction to Machine Learning for physicists. *Physics Reports*, 810:1–124, 2019.

[28] Milad Hobbi Mobarhan, Geir Halnes, Pablo Martínez-Cañada, Torkel Hafting, Marianne Fyhn, and Gaute T. Einevoll. *Firing-rate based network modeling of the dLGN circuit: Effects of cortical feedback on spatiotemporal response properties of relay cells*, volume 14. 2018.

[29] Eivind S. Norheim, John Wyller, Eilen Nordlie, and Gaute T. Einevoll. A minimal mechanistic model for temporal signal processing in the lateral geniculate nucleus. *Cognitive Neurodynamics*, 6(3):259–281, 2012.

[30] Marco Piccolino. Animal electricity and the birth of electrophysiology: The legacy of Luigi Galvani. *Brain Research Bulletin*, 46(5):381–407, 1998.

[31] R. W. Rodieck. Quantitative Analysis of Cat Retinal Ganglion Cell Response to Visual Stimuli. *Vision Research*, 5(12):573–700, 1965.

[32] Dominik Scherer, Andreas Müller, and Sven Behnke. Evaluation of pooling operations in convolutional architectures for object recognition. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 6354 LNCS(PART 3):92–101, 2010.

[33] Alex Sherstinsky. Fundamentals of Recurrent Neural Network (RNN) and Long Short-Term Memory (LSTM) Network. pages 1–39, 2018.

[34] Kyriaki Sidiropoulou, Eleftheria Kyriaki Pissadaki, and Panayiota Poirazi. Inside the brain of a neuron. *EMBO Reports*, 7(9):886–892, 2006.

[35] Scrhittweiser Julian Silver David, Hubert Thomas. Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm. *Annals of Clinical and Translational Neurology*, 2017.

[36] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A Simple Way to Prevent Neural Networks from Overfitting. *Journal of Machine Learning Research*, 15:1929–1958, 2015.

[37] David Sterratt, Bruce Graham, Computing Science, Computing Science, Andrew Gillies, Psymetrix Limited, David Willshaw, and Computational Neurobiology. *Principles of Computational Modelling in Neuroscience*.

[38] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. Intriguing properties of neural networks. pages 1–10, 2013.

[39] Peter D Welch. The Use of Fast Fourier Transform for the Estimation of Power Spectra: A Method Based on Time Aver. aging Over Short, Modified Periodograms. *IEEE Transactions on Audio and Electroacoustics*, AU-15(2):70–73, 1967.

[40] Geoffrey B. West. *Scale: The Universal Laws of Growth, Innovation, Sustainability, and the Pace of Life in Organisms, Cities, Economies, and Companies*. Penguin Books, 2018.

[41] The Making Sense Podcast with Sam Harris. Episode 86: From cells to cities, 2017.