School of Natural Sciences
Discipline of Mathematics

SPARSE CODING IN DEEP NEURAL NETWORKS AND THE VISUAL SYSTEM

by

Niam Roy Askey-Doran, BSc, BPhil.

November 2021

Submitted in partial fulfilment of the requirements for the Degree of
Bachelor of Science with Honours

Supervisors: Professor Michael Charleston
Dr William Connelly

I declare that this thesis contains no material which has been accepted for a degree or diploma by the University or any other institution, except by way of background information and duly acknowledged in the thesis, and that, to the best of my knowledge and belief, this thesis contains no material previously published or written by another person, except where due acknowledgement is made in the text of the thesis.

Signed: _____
Niam Roy Askey-Doran

Date: _____

Signed: _____
Niam Roy Askey-Doran

Date: _____

ABSTRACT

The response properties of neurons in the visual cortex of mammals has been studied for decades, and much theoretical and experimental work has been done in an attempt to understand the principles governing the way in which the visual cortex processes visual information. Many of these studies suggest that the brain follows a "sparse coding" strategy, in which visual stimuli are encoded by only a small number of active neurons at any one time. Sparse coding may be advantageous as it better represents the structure of natural scenes and is energy efficient. This strategy has previously been implemented in liner coding models to some success, replicating the response properties of simple cells in the early stages of the visual cortex. However, the role of sparse coding in later stages of the visual system is less clear, and there is still much to learn about the role of sparse coding in sensory processing.

This thesis aims to use deep neural networks as an alternative computational avenue in which to explore sparse coding. In this thesis we train fully connected neural networks, convolutional neural networks, and autoencoders on simple image datasets, and specifically penalise these networks for a lack of sparse coding. By varying this penalty and the network structure, we investigate how sparseness varies across layers of the network and how this alters the performance of the network.

This work has found that sparse and useful activity can be created in all three neural network architectures tested without significant hits to performance, and that sparsity can help to reduce redundancy in learnt features. It has also found that sparsity tends to decrease (i.e., activity increases) across layers in a deep network rather than increase. Finally, we conclude that deep neural networks are not necessarily a good cortical analogue in which to explore sparsity, but may still offer useful insights into sparsity as a coding principle.

ACKNOWLEDGEMENTS

First I would like to thank my supervisors, Michael Charleston and Bill Connelly. Thank you to Bill for originally suggesting the project, and to Michael for agreeing to take it on. Your guidance throughout the year has been essential.

Second I would also like to thank my family who have supported me throughout the year. You have allowed me the space and time I needed to complete this thesis and supported me through it all.

Finally I would like to thank my friends who, despite my neglect of them, have been there to support me every step of the way, as well as everyone else who has helped along the way. You know who you are, and without you this would not have been possible.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

Introduction

The visual system of the brain has the challenge of finding a meaningful interpretation of the environment based on sensory input to the retina. The brain must understand the elements of a 3-dimensional world using a 2-dimensional array of sensors (the retina in each eye), but the intensity of the light hitting the retina depends on multiple factors: the geometry of the object, its reflectance, its roughness, and the properties of the light source. Determining which of these properties are the true explanations of an object's appearance based on the information received by the retina is a underdetermined problem with no unique solution [1]. The information collected by the retinas is not enough to entirely explain the organism's environment. How then does the brain make sense of the world it sees?

A popular theory of visual perception is that in order to tackle this problem the visual cortex contains its own probabilistic model of the environment [2, 3, 4, 5]. There may be many possible explanations for the visual signals received by the retina, however some explanations are more likely than others. It is theorised that the brain models these probabilities, incorporating its knowledge of the environment and past experience of which explanations are most memorable, and *infers* the most probable reality.

Ideas have previously been taken from information theory and statistics in an attempt to understand the principles underlying the way in which the brain forms this probabilistic model, and how it encodes this information. One possible principle that the primary visual cortex (V1) appears to follow is the principle of *sparse coding*. The pattern of neural activity that results from a given stimulus is called the neural *code*. A *sparse* neural code is one in which a stimulus is encoded by the activity of a small proportion of neurons in a population, while the majority of neurons in the population remain silent. Mathematically, images can be decomposed into a series of basis functions - simpler "building blocks" that can be used to construct more complicated signals such as images. In the language of signal processing, a sparse code is one in which a signal (in this case an image) is represented as accurately as possible using the fewest basis functions.

The role of sparse coding in the visual system and perception has previously been explored in linear coding models, and this work has been successful in explaining some properties of neurons in the early levels of the visual cortex [2, 5, 6, 7]. However, subsequent layers of the visual system are less understood, as is the role of sparsity in these regions. The popularisation of neural network based machine learning models, and the ease with which they can be created using packages such as PyTorch and Keras [8, 9], offers an alternative computational avenue in which sparseness can be investigated. While neural networks are a gross oversimplification of the biological reality, they have an advantage over past models in that deep, multi-layered networks can be created in which the role of sparsity across a hierarchical structure can be studied. This thesis aims to enforce sparse activation in a variety of deep neural network architectures in order to explore whether sparse coding benefits or hinders the performance of neural networks, and to see how enforcing sparsity in a multi-layered network alters activity across various layers. By doing so we hope to gain a better understanding of the role of sparse coding both in the brain and in learning algorithms such as neural networks.

1.1 Background

1.1.1 The Visual System

The visual system can be roughly divided into separate functional regions, beginning at the eyes. Rods and cones on the retina respond to incident light and become excited, sending signals off to neighbouring cells. Signal processing begins in the retina by a variety of these neighbouring cells. This information is then sent into the brain to the lateral geniculate nucleus (LGN) via the optic nerve. The LGN acts as a processing station, receiving information not only from the retinas but from the cortex and brainstem as well, before sending this information off to V1 in the visual cortex [10]. The majority of visual perception occurs in the visual cortex, which consists of multiple regions (labelled V1 - V5) separated by function and structure. V2 receives inputs from V1, and outputs both feedback projections to V1 and feedforward connections to the remaining cortical regions (V3, V4, and V5) [11].

1.1.2 Early Theories of Visual Perception

Early experimental work focused on measuring the response properties of individual neurons to a given stimuli at various stages throughout the visual stream [12, 13, 14]. Images are formed by light reflecting off objects in the world and entering the eye, where it stimulates an array of photoreceptors on the retina. The response properties of a neuron can partly be described by the neurons *receptive field* - the location, motion, and pattern of light on the retina to which a neuron will respond. Receptive fields can

consist of "on" regions, where incident light will cause the neuron to become excited, and "off" regions, where incident light will instead inhibit the neuron.

It was understood from this early work that retinal ganglion cells (RGCs), which receive input from nearby photoreceptors in the retina via intermediate cells, exhibit so called "centre-surround" receptive fields. These are circular receptive fields consisting of central and surrounding regions with opposite responses to light, i.e., a central "on" region surrounded by an "off" region, or *vice versa* [12]. For example, if light is shone only on the "on" region of the receptive field, the RGC will become excited above its basal frequency (the rate at which it activates without any stimulus). Conversely if light is shone only on the "off" regions, the RGC will become inhibited. If light is shone on the entire receptive field, the RGC will fire at a rate somewhere between these two extremes.

Further work by Hubel and Weisel soon demonstrated that neurons in the cat primary visual cortex exhibit more complicated receptive fields than those demonstrated by ganglion cells [13]. Instead of exhibiting circular receptive fields, neurons in V1 responded to oriented, elongated bands of light. Such receptive fields are often described as spatially localised, oriented, and bandpass (i.e., selective to particular spatial frequencies) [6]. These bands again followed a pattern consisting of a central "on" band flanked by "off" regions on one or both sides, or vice versa. Cells with such a receptive field were termed *simple cells* [10]. Another class of cells called *complex cells* were also found in V1. These cells in general do not contain "on" and "off" regions, are highly orientation-selective, and respond to bars of oriented light at *any* location in their receptive field (i.e., are not spatially-selective) [10]. This work led to the idea that neurons in successive layers of the visual cortex have receptive fields built from the receptive fields of previous layers [14]. For example, the elongated receptive fields of simple cells in V1 may be constructed by combining the inputs from a series of RGCs whose receptive fields form a line. Similarly, complex cells combine the inputs from a set of simple cells that respond to the same orientation of light. In this way each successive area of the visual system constructs more and more complicated receptive fields from the inputs of simpler cells.

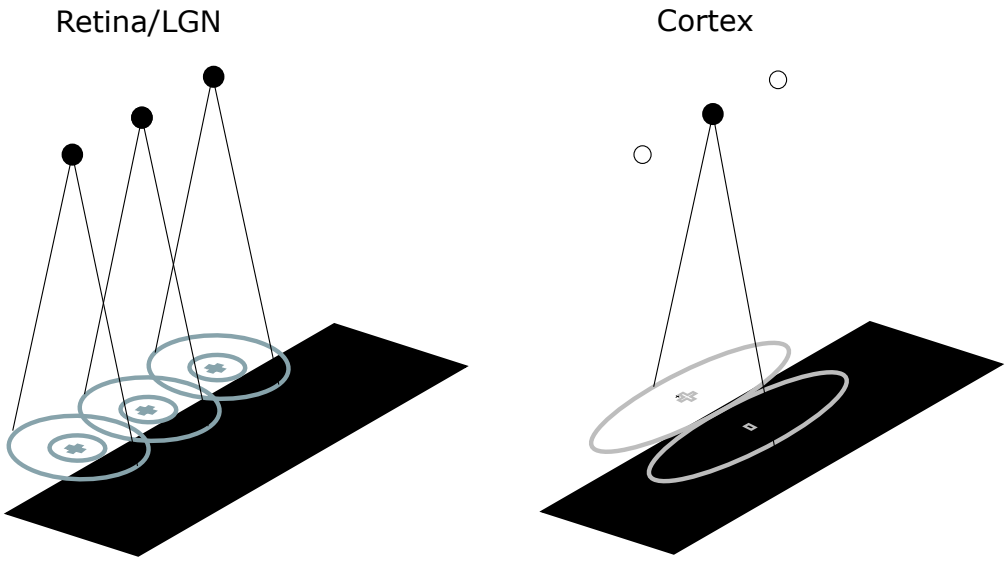


Figure 1.1: Cortical neurons can build their receptive fields by combining those of neurons in earlier layers. Image taken from Olshausen 2003 [3]

How then does the brain "decide" what receptive field a neuron should have? The simplicity of the model described above soon became unpopular as the use of information theory and statistics grew among those researching visual perception, and researchers were suggesting that it was not enough to only focus on the computations of individual neurons [5]. Ideas from information theory were also being proposed as guiding principles for the way in which the visual system processes sensory information. Attneave suggested that the brain seeks to take advantage of the statistical redundancies present in visual scenes [15]. Images contain redundant information in both space and time - given knowledge about part of an object it is often possible to deduce the structure of the entire object by taking advantage of any symmetries, homogeneities, or patterns in the object. Barlow codified this idea as "redundancy reduction", with the theory being that the brain seeks an efficient code that minimises redundant information [16, 17].

Since then, a particular emphasis has been placed on perception as a process of statistical inference. Barlow argued that, "instead of thinking of neural representations as transformations of stimulus energies, we should regard them as approximate estimates of the probable truths of hypothesis about the current environment" [4]. In other words, the brain doesn't just "calculate" an explanation of its environment by transforming and filtering the signals it receives - rather it makes an "educated guess" based on its prior knowledge of the world and the signals it receives. In order to perform this visual inference, the brain forms a probabilistic model of its environment which describes the probabilities of observing a stimulus, and the distribution of possible explanations for these stimuli.

The way in which the visual system evolves to respond to stimuli should be driven by three factors: (1) the tasks that the organism must perform, (2) the biological limitations

of the visual system, and (3) the environment in which the organism lives [18]. Thus the brain must develop a meaningful code which represents the statistical properties of its environment and maximises chances of survival, subject to the biological constraints of the system. Considering these factors, and incorporating the idea of perception as a process of inference, Olshausen and Field suggested that the brain aims to find a *sparse* representation of visual stimuli. It has been suggested that natural scenes – typical images and scenes we might see every day – are sparse in structure [19], and so a sparse neural code would be capable of reflecting the structure of the environment. A sparse code also minimises the number of elements used to encode a signal. This is a significant advantage for a biological organism as brains are expensive to run, and minimising energy expenditure increases an organisms chance of survival. An estimated 20,500 ATP molecules (equivalent to roughly 2.05×10^{-19} joules) are required for the transmission of one bit of information [20]. Sparse coding, however, decreases the metabolic cost of perception [7]. The idea that the visual cortex employs a sparse coding scheme is also supported by experimental evidence [21, 22, 23, 24].

1.1.3 Sparse Neural Codes

One could imagine a number of patterns of neural activity that could occur in response to a particular stimulus. Which particular neurons respond to a stimulus, and the way in which they respond, constitutes a *neural code* - i.e. the information of the stimulus is somehow represented by the activity of these neurons.

Consider the population of neurons that make up the primary visual cortex. One extreme possibility is that all of these neurons fire to varying degrees in response to visual stimuli. In this case, some neurons would respond more intensely than others for a given stimulus, but all neurons fire to some degree. This is called a *dense population code*. Such a coding regime could be robust to noisy inputs. They would also have a high representational capacity, allowing up to M^N contexts to be encoded by N neurons with M activity levels [25]. However it would be energetically costly, and would potentially be low fidelity - if all neurons are firing in response to every stimulus then it is more difficult to differentiate between different neural codes, causing different stimuli to "blur" together [2].

Another extreme is to ultimately end up with a single neuron firing in response to a given stimulus. In the most extreme case this would mean that every object the organism might see has a corresponding neuron that only fires when that object is seen. This is often called a *local code* [2, 25]. While much cheaper energetically, such a coding regime would require an unreasonable amount of neurons in the visual cortex to encode every possible visual stimulus (since N neurons could only encode N contexts [25]).

Sparse coding exists in the "sweet spot" between these two extremes. A sparse neural code is simply one in which a relatively small proportion of neurons within a population are used to represent a given stimulus [2, 25]. Such a code is still a population code, as

the work of representing a stimulus is distributed among neurons in a group, however for any particular stimulus only a small proportion of neurons need be active.

1.2 Linear Codes and Sparsity

Finding efficient and useful methods of encoding signals and information is a common problem in signal processing and engineering. Commonly, the goal is to find a low-dimensional representation of a signal in order to compress the information it contains. One approach taken to explore ideas about visual processing in the brain has been to find algorithms to "learn" such representations of images, and ensure that the algorithm follows the principle being studied. The learnt representation can then be compared to the receptive fields of neurons in the visual cortex. Here we introduce the idea of linear coding and sparse linear codes, and how these can be used to explore sparse coding in the visual cortex.

1.2.1 Linear Codes

Linear codes aim to represent an input as a linear combination of basis functions. Visual stimuli (images) are 2-dimensional inputs, so the linear code of an image has the form

$$\hat{I}(x,y) = \sum_i a_i \phi_i(x,y) \tag{1.1}$$

where $\phi_i(x,y)$ are the basis functions, a_i are the coefficients, $\hat{I}(x,y)$ is the reconstruction of the image $I(x,y)$ being coded, and x and y denote the coordinates within the image. Models such as this are termed *generative models* [2].

Greyscale images are represented by 2D arrays of pixel values. When considering the linear code of an image, the basis functions $\phi_i(x,y)$ are also arrays with the same dimension as the image. In practice these are often flattened into a vector, $\underset{\sim}{\boldsymbol{\phi_i}}$ and placed into a matrix, $\boldsymbol{\Phi}$, where the columns correspond to each flattened basis function. This allows Equation 1.1 to alternatively be written in vector form as

$$\underset{\sim}{\hat{\boldsymbol{I}}} = \boldsymbol{\Phi}\underset{\sim}{\boldsymbol{a}}, \tag{1.2}$$

where $\underset{\sim}{\hat{\boldsymbol{I}}}$ is now also a vector representing the flattened version of the reconstructed image. $\boldsymbol{\Phi}$ is often called a *dictionary*, and contains all the basis functions available to encode an image. The dictionary may be chosen from a well known basis if the properties of the signals being encoded are well known (e.g., Fourier or wavelet basis functions), or as in this case may be learnt via some learning algorithm. Basis functions need not be orthogonal, although many learning algorithms do find orthogonal basis sets, however the dictionary should span the space of inputs. The components of the coefficient vector $\underset{\sim}{\boldsymbol{a}}$ determine which basis functions are used.

In neural coding models, the dictionary is often chosen to be *overcomplete*. A dictionary is overcomplete if the number of functions in the set is larger than the dimensionality of the input. By allowing overcompleteness, the model has a greater flexibility in matching the structure of input data [2]. Allowing overcompleteness is also biologically realistic – there are far more neurons in the V1 than there are inputs into it from the LGN (by about 500:1), suggesting that the representational capacity of V1 is overcomplete [7].

The problem of finding a linear code can be framed as an optimisation problem, where the goal is to minimise an objective function J with respect to both $\underset{\sim}{\boldsymbol{a}}$ and $\boldsymbol{\Phi}$. The objective function offers some measure of how well a chosen set of bases and coefficients represent the original data. The mean square error is a common objective function,

$$J = \|\underset{\sim}{\boldsymbol{I}} - \boldsymbol{\Phi}\underset{\sim}{\boldsymbol{a}}\|_2^2 = \sum_{x,y}\left[I(x,y) - \sum_i a_i\phi_i(x,y)\right]^2 \tag{1.3}$$

where $\|\cdot\|_2$ is the ℓ_2 norm. Hence the goal is to find a solution to the following:

$$\hat{\underset{\sim}{\boldsymbol{a}}}, \hat{\boldsymbol{\Phi}} = \underset{\underset{\sim}{\boldsymbol{a}},\boldsymbol{\Phi}}{\operatorname{argmin}}\left\{\|I - \boldsymbol{\Phi}\underset{\sim}{\boldsymbol{a}}\|_2^2\right\}, \tag{1.4}$$

where $\hat{\underset{\sim}{\boldsymbol{a}}}$ and $\hat{\boldsymbol{\Phi}}$ represent the optimal solution set. The objective function can be minimised with the use of a learning algorithm. The goal of a linear coding learning algorithm is twofold: to learn a dictionary of basis functions from the data that can best describe the range of inputs (if we do not already know what the most suitable basis functions for the job are) and, given this set of basis functions, find the optimal representation of a given input using elements from the learnt dictionary.

When considering models of neural coding, the basis functions $\phi_i(x,y)$ are considered analogous to the receptive fields of a neuron, and the coefficients a_i are considered analogous to the activation of the i^{th} neuron. Linear coding models can often be framed as artificial neural networks, in which case the coefficients correspond to the activation value of the nodes in the network, and the basis functions correspond to the set of weights leading into each node (and may also be called receptive fields). These terms may be used interchangeably. In deep networks however (see Section 1.4), the notion of a receptive field changes slightly, and the weights leading directly into a node may not always correspond to that node's receptive field. Table 1.1 shows a comparison of these terms.

Table 1.1: Terms used in models of neural coding.

Term	Brain	Coding Models	Neural Networks
a_i	Neuron activity	Coefficients	Node/Neuron activation value
ϕ_i	Receptive field	Basis functions	Receptive field/Node weights

1.2.2 Sparse Linear Codes

A strict definition of what constitutes a sparse code is often not articulated in the literature, and different contexts may require slightly different definitions. Ultimately the best definition depends on the context and the reasons for desiring sparsity, and a more comprehensive understanding is gained by using multiple measures.

An intuitive definition of a sparse code in this context is one in which only a small proportion of the available basis functions are used to represent a given image. Equivalently, a sparse code is one for which most of the entries in the vector of coefficients, $\underset{\sim}{\boldsymbol{a}}$, are zero. $\underset{\sim}{\boldsymbol{a}}$ is called K-sparse in $\boldsymbol{\Phi}$ if there are exactly K non zero elements in $\underset{\sim}{\boldsymbol{a}}$ [26].

Much like the problem of visual perception that the visual cortex must solve, minimising Equation 1.3 with respect to both $\underset{\sim}{\boldsymbol{a}}$ and $\boldsymbol{\Phi}$ is under-determined and has no unique solution [27]. It is necessary then to impose some restrictions on the form of the linear code – we must make an assumption about what kind of linear code will best describe our data. It is here that we assume that a sparse code is the most appropriate form of linear code. A sparse code can be formed by adding a *regularisation* term, or function, $S(\underset{\sim}{\boldsymbol{a}})$, to the objective function. The role of the regularisation term is to add an extra cost for every non-zero coefficient used in the code. Regularisation turns the problem of minimising J into a constrained optimisation problem, and may be framed in a couple of equivalent ways.

Regularisation as an Optimisation Constraint

Regularisation can be seen as turning Equation 1.4 into a *constrained* optimisation problem. Let $S(\underset{\sim}{\boldsymbol{a}})$ be the regularisation term which, in our case, gives some measure of the sparsity of the coefficients of the code. The constrained optimisation problem can then be stated as

$$\hat{\underset{\sim}{\boldsymbol{a}}}, \hat{\boldsymbol{\Phi}} = \underset{\underset{\sim}{\boldsymbol{a}}, \boldsymbol{\Phi}}{\operatorname{argmin}} \left\{ \|I - \boldsymbol{\Phi}\underset{\sim}{\boldsymbol{a}}\|_2^2 \right\} \quad \text{subject to } S(\underset{\sim}{\boldsymbol{a}}) \ \leq t \tag{1.5}$$

where t is some minimum level of sparsity we wish to achieve. This may equivalently be solved by the method of Lagrange multipliers (by Lagrange duality) [28, 29]:

$$\hat{\underset{\sim}{\boldsymbol{a}}}, \hat{\boldsymbol{\Phi}} = \underset{\underset{\sim}{\boldsymbol{a}}, \boldsymbol{\Phi}}{\operatorname{argmin}} \left\{ \|I - \boldsymbol{\Phi}\underset{\sim}{\boldsymbol{a}}\|_2^2 + \lambda S(\underset{\sim}{\boldsymbol{a}}) \right\}. \tag{1.6}$$

Constrained optimisation using this method is a well known problem, and a range of solution methods exist depending on the choice of $S(\underset{\sim}{\boldsymbol{a}})$ [27]. Considering regularisation as an optimisation constraint is useful in the context of neural networks.

Regularisation as a Bayesian Prior

Alternatively, regularisation may be imposed as a Bayesian prior over the coefficients. Suppose we have an image, I, for which we wish to find a linear code, and a dictionary of basis functions $\mathbf{\Phi}$ (i.e., assuming we already know which basis best describes our data). As stated before the problem of finding a linear code has no unique solution, so given an image and a dictionary there is a distribution of coefficients that might encode the image, $P(\underset{\sim}{\boldsymbol{a}}|I, \mathbf{\Phi})$. This may be calculated via Bayes' rule,

$$P(\underset{\sim}{\boldsymbol{a}}|I, \mathbf{\Phi}) \propto P(I|\underset{\sim}{\boldsymbol{a}}, \mathbf{\Phi})P(\underset{\sim}{\boldsymbol{a}}),$$

where $P(I|\underset{\sim}{\boldsymbol{a}}, \phi)$ is the likelihood of an image arising from a given dictionary and set of coefficients, and $P(\underset{\sim}{\boldsymbol{a}})$ is the prior distribution of coefficients. As the role of the coefficients is to select which basis functions are used to describe an image, $P(\underset{\sim}{\boldsymbol{a}})$ represents our assumptions about the structure of our data and how it is best described.

By maximising $P(\underset{\sim}{\boldsymbol{a}}|I, \phi)$ we can find the coefficients that best describe our image. This is done through Maximum *A Posteriori* (MAP) estimation:

$$\hat{\underset{\sim}{\boldsymbol{a}}} = \underset{\underset{\sim}{\boldsymbol{a}}}{\operatorname{argmax}}\left\{\log(P(\underset{\sim}{\boldsymbol{a}}|I, \mathbf{\Phi}))\right\} = \underset{\underset{\sim}{\boldsymbol{a}}}{\operatorname{argmax}}\left\{\log(P(I|\underset{\sim}{\boldsymbol{a}}, \mathbf{\Phi})) + \log(P(\underset{\sim}{\boldsymbol{a}}))\right\} \quad (1.7)$$

Now, we again assume a linear coding model, but also assume that there is some noise, ν, in the coding process

$$\hat{I}(x, y) = \sum_i a_i \phi_i(x, y) + \nu(x, y).$$

This noise can be modelled as a Gaussian function, as we assume that the coding model can account for all explanations of the image except for those which are random and independent, which combine in the limit to follow a Gaussian [5]. The likelihood of the model is then given as

$$P(I|\underset{\sim}{\boldsymbol{a}}, \mathbf{\Phi}) \propto \exp \frac{-\|I - \mathbf{\Phi}\underset{\sim}{\boldsymbol{a}}\|_2^2}{2\sigma_n^2}, \quad (1.8)$$

where σ_n^2 is the variance of the noise. Since we wish to find a sparse code, the prior distribution over the coefficients must be chosen such that the probability of a coefficient being zero is high. The prior distribution can be given as

$$P(\underset{\sim}{\boldsymbol{a}}) \propto e^{-\lambda S(\underset{\sim}{\boldsymbol{a}})}, \quad (1.9)$$

where $S(\underset{\sim}{\boldsymbol{a}})$ is again our measurement of sparsity. Given these distributions, the MAP estimation becomes

$$\begin{aligned}\hat{\underset{\sim}{\boldsymbol{a}}} &= \underset{\underset{\sim}{\boldsymbol{a}}}{\operatorname{argmax}}\left\{-\frac{1}{2\sigma_n^2}\|I - \mathbf{\Phi}\underset{\sim}{\boldsymbol{a}}\|_2^2 - \lambda S(\underset{\sim}{\boldsymbol{a}})\right\} \\ &= \underset{\underset{\sim}{\boldsymbol{a}}}{\operatorname{argmin}}\left\{\frac{1}{2\sigma_n^2}\|I - \mathbf{\Phi}\underset{\sim}{\boldsymbol{a}}\|_2^2 + \lambda S(\underset{\sim}{\boldsymbol{a}})\right\} \quad (1.10)\end{aligned}$$

which is again the problem in Equation 1.6 (although here we have assumed we already know $\mathbf{\Phi}$, but this makes no difference).

Viewing regularisation as a Bayesian prior is useful if, as in this case, one knows the desirable properties of an optimal linear code. This also gives a more intuitive connection between the linear coding models presented in this section, and the way in which the visual cortex itself may be inferring its environment. In this case, the coefficients can be thought of as the elements of an environment, or the "explanations" for a given image. The image can be though of as the data - the image seen by the visual system is the pattern in which the photoreceptors in the retina activate, and this is the data the visual cortex must use to infer the structure of its environment. So, given a model for how the environment, E, gives rise to a given pattern of photoreceptor activations in the eyes, D, and a *prior* distribution describing how likely given states of the environment are, $P(E)$, the brain may infer its environment from a set of observed data via Baye's rule:

$$P(E|D) \propto P(D|E)P(E)$$

This method is commonly used in the literature when deriving sparse coding models of the visual cortex [2, 5, 6]. A number of sparsity metrics exist which can be used both as the regularisation function $S(\underset{\sim}{\boldsymbol{a}})$ and as a measure of how sparse a code is [30].

1.2.3 Sparse Regularisation Functions

A variety of sparsity measures exist that may be used for various contexts [30]. The ℓ_0 pseudo-norm is a popular and seemingly natural choice of regularisation function. The ℓ_0 norm gives the number of non-zero elements of a vector, $\underset{\sim}{\boldsymbol{x}}$, and is defined as the limit as $p \to 0$ of the ℓ_p norm [27]. The ℓ_0 norm is a good choice of sparsity measure in this case, as we are interested in codes in which most coefficients are zero.

Definition 1.2.1. *(ℓ_0 pseudo-norm)*

$$\|\underset{\sim}{\boldsymbol{x}}\|_0 = \lim_{p\to 0} \|\underset{\sim}{\boldsymbol{x}}\|_p = \lim_{p\to 0} \Big(\sum_{i=1}^{n} |x_i|^p\Big)^{\frac{1}{p}} \tag{1.11}$$

where $\|\underset{\sim}{\boldsymbol{x}}\|_p = \left(\sum_{i=1}^{n} |x_i|^p\right)^{\frac{1}{p}}$ is the ℓ_p norm, and n is the number of elements in the vector $\underset{\sim}{\boldsymbol{x}}$. Since the ℓ_0 norm counts the number of non-zero elements in $\underset{\sim}{\boldsymbol{x}}$, it may be expressed as:

$$\|\underset{\sim}{\boldsymbol{x}}\|_0 = \sum_{i=1}^{n} t_i, \quad \text{where } t_i = \begin{cases} 1, & x_i > 0 \\ 0, & \text{otherwise} \end{cases} \tag{1.12}$$

Solving Equation 1.6 with $S(\underset{\sim}{\boldsymbol{a}}) = \|\underset{\sim}{\boldsymbol{a}}\|_0$ is unfortunately an NP-hard problem, in part due to the non-differentiability of the ℓ_0 norm [27, 30]. An exact solution would require an exhaustive search over every subset of basis functions $\mathbf{\Phi}$ [31] – a combinatorial

nightmare. Strategies do however exist to solve this problem approximately, such as the matching pursuit algorithm [27].

Matching pursuit is a greedy strategy to approximately solve Equation 1.6 with $S(\underset{\sim}{\boldsymbol{a}}) = \|\underset{\sim}{\boldsymbol{a}}\|_0$, and was one of the earliest methods to do so [27]. It involves iteratively choosing a basis function from the dictionary for which the inner product between the basis function and the image *residual* is maximum, where the residual represents the difference between the image being coded and the current image code. For an iteration i, a basis function is selected such that

$$|\langle \underset{\sim}{\boldsymbol{r}}^{(i)}, \underset{\sim}{\boldsymbol{\phi}}^{(i)} \rangle| = \sup \left\{ |\langle \underset{\sim}{\boldsymbol{r}}^{(i)}, \underset{\sim}{\boldsymbol{\phi}}_j \rangle| \right\} \tag{1.13}$$

where $\underset{\sim}{\boldsymbol{\phi}}_i$ is an element of the dictionary, $\underset{\sim}{\boldsymbol{\phi}}^{(i)}$ is the basis function chosen in the i^{th} iteration, $r^{(i)}$ is the i^{th} residual, and $\langle \underset{\sim}{\boldsymbol{x}}, \underset{\sim}{\boldsymbol{y}} \rangle$ is the inner product between two vectors. For the first iteration, the residual is given as

$$\underset{\sim}{\boldsymbol{r}}^{(0)} = I = \langle I, \underset{\sim}{\boldsymbol{\phi}}^{(0)} \rangle \underset{\sim}{\boldsymbol{\phi}}^{(0)} + \underset{\sim}{\boldsymbol{r}}^{(1)} \tag{1.14}$$

and for iteration i,

$$\underset{\sim}{\boldsymbol{r}}^{(i)} = \langle \underset{\sim}{\boldsymbol{r}}^{(i)}, \underset{\sim}{\boldsymbol{\phi}}^{(i)} \rangle \underset{\sim}{\boldsymbol{\phi}}^{(i)} + \underset{\sim}{\boldsymbol{r}}^{(i+1)}. \tag{1.15}$$

This way the linear code for the image may be constructed by iteratively selecting elements until the residual is sufficiently small.

Another popular choice of regularisation is the ℓ_1 norm. The ℓ_1 norm approximates the ℓ_0 norm [30], and under certain conditions minimising Equation 1.6 with $S(\underset{\sim}{\boldsymbol{a}}) = \|\underset{\sim}{\boldsymbol{a}}\|_1$ is exactly equivalent to using the ℓ_0 norm [31]. The ℓ_1 norm has the advantage of being convex and differentiable everywhere but at the origin. While the ℓ_1 norm may work well as a regularisation function, it may not be a good measure of sparsity in this context as it will simply measure the sum of all coefficients in the code. A code with a few large coefficients could look just as sparse as a code with many small coefficients.

Definition 1.2.2. *(ℓ_1 norm)*

$$\|\underset{\sim}{\boldsymbol{x}}\|_1 = \sum_i |x_i| \tag{1.16}$$

The process of solving Equation 1.6 with $S(\underset{\sim}{\boldsymbol{a}}) = \|\underset{\sim}{\boldsymbol{a}}\|_1$ is called basis pursuit denoising [27, 31], and may be solved via a number of algorithms such as the Iterative Shrinkage Thresholding Algorithm (ISTA) [27], or Alternating Projected Gradient Descent [5].

Gradient descent describes a family of optimisation algorithms that minimise a function by iteratively changing the parameters of the objective function by an amount proportional to the gradient of the cost function. In its simplest form, the parameters are updated each iteration as

$$\underset{\sim}{\boldsymbol{a}}^{new} = \underset{\sim}{\boldsymbol{a}}_i^{old} - \alpha \nabla J(\underset{\sim}{\boldsymbol{a}}, \underset{\sim}{\boldsymbol{\phi}}) \tag{1.17}$$

where α is a proportionality constant called the *learning rate*.

In alternating gradient descent, this process is simply performed iteratively on one parameter at a time, or by alternating between two sets of parameters in J (for example in this case one would alternate between optimising with respect to the coefficients, and optimising with respect to the basis functions). Projected gradient descent is used when a constraint is involved, and involves first taking a gradient descent step, then projecting the result onto the constraint set [28].

Other measures exist such as the Hoyer measure and the Gini index [30]. These are normalised measures, and so measure sparsity as a value between 0 and 1. The Gini index was originally developed as a measure of inequality of wealth, and weights coefficients by size so that changing a small coefficient will impact sparsity more than changing a large coefficient. The Gini index is defined as

Definition 1.2.3. *(Gini Index)*

$$G(\underset{\sim}{\boldsymbol{a}}) = 1 - 2\sum_{k=1}^{N} \frac{a_k}{\|\underset{\sim}{\boldsymbol{a}}\|_1}\left(\frac{N-k+\frac{1}{2}}{N}\right), \tag{1.18}$$

where N is the number of elements in $\underset{\sim}{\boldsymbol{a}}$, and the elements of $\underset{\sim}{\boldsymbol{a}}$ are ordered from smallest to largest so that $a_1 < a_2 < \cdots < a_N$, where $1, 2, \ldots, N$ are the indices of the ordered list, and a_k is an element of the ordered list.

Hoyer's index is based on the ratio of the ℓ_1 and ℓ_2 norms, and is also a normalized measure of sparsity.

Definition 1.2.4. *(Hoyer's Index)*

$$H(\underset{\sim}{\boldsymbol{a}}) = \left(\sqrt{N} - \frac{\sum_j a_j}{\sqrt{\sum_j a_j^2}}\right)(\sqrt{N}-1)^{-1} \tag{1.19}$$

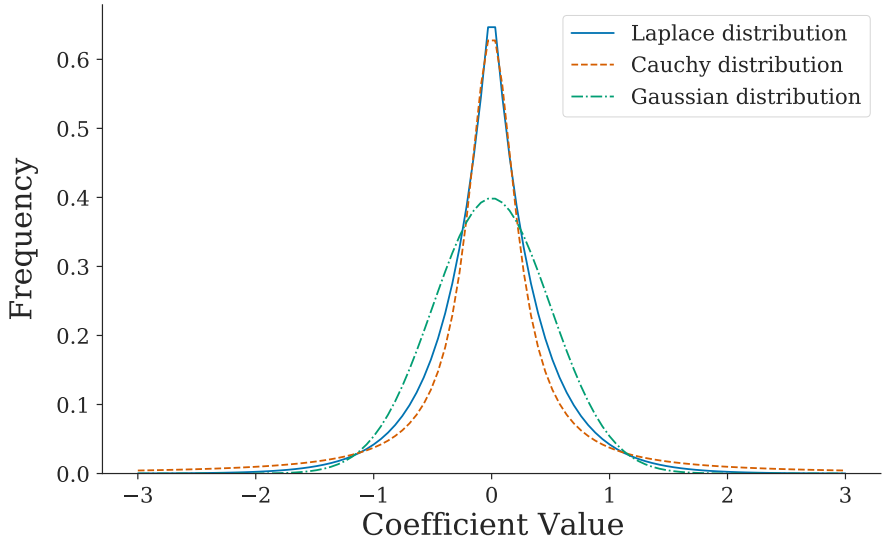
Forms of sparsity enforced by a continuous regularisation function are sometimes called "soft" forms, and generally lead to a distribution of coefficients that is more peaked than a Gaussian [2, 7]. Suitable soft forms of $S(\underset{\sim}{\boldsymbol{a}})$ are $S(\underset{\sim}{\boldsymbol{a}}) = \|\underset{\sim}{\boldsymbol{a}}\|_1$, which leads to a Laplace distribution of coefficients, or $S(\underset{\sim}{\boldsymbol{a}}) = \sum_i \log(1+a_i^2)$, which leads to a Cauchy distribution of coefficients (when $\lambda = 1$) [2], shown in Figure 1.2a.

This correspondence between the choice of $S(\underset{\sim}{\boldsymbol{a}})$ and the coefficient distributions can be derived by considering regularisation as a Bayesian prior as above. If $S(\underset{\sim}{\boldsymbol{a}}) = \|\underset{\sim}{\boldsymbol{a}}\|_1$, the prior distribution of values for a single coefficient becomes

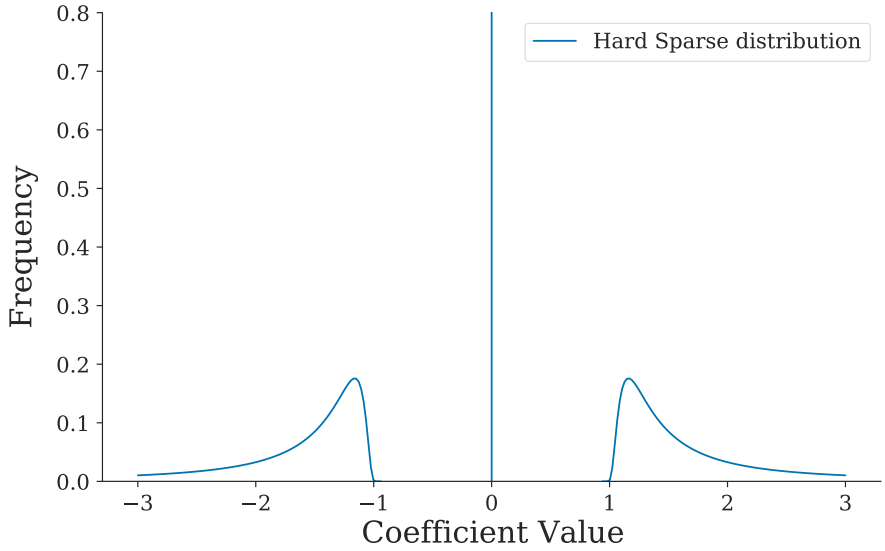
$$P(a_i) \propto e^{-\lambda|a_i|}$$

which is a Laplacian distribution (where any other constants have either been absorbed by λ, or make no difference to the process of learning linear codes).

Sparseness enforced with the ℓ_0 norm is sometimes called "hard" sparseness, and leads to a discontinuous distribution of coefficients with a Dirac peak at zero, as shown in Figure 1.2b [7].



(a) Soft sparseness



(b) Hard sparseness

Figure 1.2: (a) Distribution of coefficients for various soft forms of sparse linear codes. The Laplace distribution is given by $f(x) = \frac{1}{2}e^{-|x|}$, and the Cauchy distribution is given by $f(x) = \dfrac{1}{\pi(1+x^2)}$. (b) Distribution of coefficients for hard forms of linear codes, i.e., those enforced by the ℓ_0 norm (this is simply a depiction of the distribution empirically determined by Rehn and Sommer [7]).

1.3 Sparse Coding Models

1.3.1 Sparsenet

In a landmark paper, published 1996, Olshausen and Field introduced a learning algorithm designed to find sparse linear codes for natural images [2, 6]. In particular, the model uses "soft" forms of the regularisation function, $S(\underset{\sim}{\boldsymbol{a}})$. Sparsenet finds the set of basis functions $\boldsymbol{\Phi}$ and coefficients $\underset{\sim}{\boldsymbol{a}}$ that solve the optimisation problem presented in Equation 1.5 for a set of images:

$$\hat{\underset{\sim}{\boldsymbol{a}}}, \hat{\boldsymbol{\Phi}} = \underset{\underset{\sim}{\boldsymbol{a}}, \boldsymbol{\Phi}}{\operatorname{argmin}}(J) \tag{1.20}$$

where

$$J = \underbrace{\left\| \underset{\sim}{\boldsymbol{I}} - \boldsymbol{\Phi}\underset{\sim}{\boldsymbol{a}} \right\|_2^2}_{\text{Preserve information}} + \underbrace{\lambda \sum_{i=1}^{M} S(a_i)}_{\text{Limit activity}} \tag{1.21}$$

The cost function J contains two components. The first term is generic in unsupervised learning algorithms, and defines the mean square error between the original input image and the linear code representing it. A good linear code minimises this reconstruction error.

The second term, involving $S(a_i)$, is the *regularisation term* that penalises the model for activity (i.e., for having non-zero coefficients).

The optimal set of coefficients and corresponding basis functions are found by minimising the cost function with respect to both a and ϕ. This can be seen as a "tug-of-war" between preserving as much information as possible in the final representation of the input, while using as few elements as possible to describe the input.

Olshausen and Field solved Equation 1.20 via alternating projected gradient descent [5]. As described earlier, in alternating gradient descent a multivariate objective function is optimised one variable at a time – in this case alternating between optimising J with respect to the coefficients, and optimising J with respect to the basis functions. This process is continued until optimisation is achieved (according to some optimality condition) using a large set of training images.

At the start of training Olshausen and Field randomly initialised the basis functions. For each training image presented to the algorithm, the coefficients were first initialised as

$$a_i^{(0)} = \langle \underset{\sim}{\boldsymbol{\phi}}_i \, , \, \underset{\sim}{\boldsymbol{I}} \rangle$$

where $\langle \cdot \rangle$ in this case denotes the dot product. The optimal coefficients were then found using an appropriate gradient descent method, iterating a_i with the following gradient [2, 6]:

$$\frac{\partial J}{\partial a_i} = a_i^{(0)} - \sum_{j \neq i} C_{ij} a_j - \lambda S'(a_i), \tag{1.22}$$

Where $a_i^{(0)} = \langle \underset{\sim}{\boldsymbol{\phi}}_i, \underset{\sim}{\boldsymbol{I}} \rangle$, and $C_{ij} = \langle \underset{\sim}{\boldsymbol{\phi}}_i, \underset{\sim}{\boldsymbol{\phi}}_j \rangle$.

Written in this form we can see two processes that are analogous to potential biological mechanisms leading to sparsity. First, there is competition between coefficients with similar receptive fields expressed as C_{ij}, the inner product between the receptive fields of two coefficients. This can be thought of as inhibition between neurons – for a code to be sparse, there should not be two active neurons with similar receptive fields. Instead, the receptive fields of each neuron should be as different as possible to maximise the information per neuron. Second, each coefficient self-inhibits via the derivative of the regularisation term, $S'(a_i)$. This pushes the value of the coefficients either towards zero, or towards a large value.

We may also rearrange Equation 1.22 as follows:

$$\frac{\partial J}{\partial a_i} = \langle \underset{\sim}{\boldsymbol{\phi}}_i, \left[\underset{\sim}{\boldsymbol{I}} - \boldsymbol{\Phi} \underset{\sim}{\boldsymbol{a}} \right] \rangle - \lambda S'(a_i). \tag{1.23}$$

When written this way we can see that each coefficient is also driven by the accuracy of the reconstructed image via the residual $\underset{\sim}{\boldsymbol{I}} - \boldsymbol{\Phi} \underset{\sim}{\boldsymbol{a}}$, weighted by its basis function. This again emphasises how the algorithm is attempting to maximise the information represented by each coefficient. If a coefficient's basis function "shares" a large amount of information with the residual term, then including this basis function in the linear code would be an efficient use of basis functions, since the residual represents information that is yet to be explained. In that case the inner product between the basis function and the residual would close to 1, and the coefficient would then be driven towards a similar value by the regularisation term in order to minimise Equation 1.23. Otherwise the inner product would be small, and the sparsity term would drive the corresponding coefficient towards a similarly small value. Note also the similarity between the first term in Equation 1.22 and the update rule for matching pursuit in Equation 1.15. The result is that the algorithm sparsifies the distribution of activities, and then learns a set of basis functions that can tolerate such a sparse set of coefficients [2].

The optimal set of basis functions is similarly found via gradient descent on J with the following update rule:

$$\Delta \phi_{ij} \propto \frac{\partial J}{\partial \phi_{ij}} = \eta a_j \left(I - \boldsymbol{\Phi} \underset{\sim}{\boldsymbol{a}} \right) \tag{1.24}$$

where ϕ_{ij} is the ij^{th} entry in $\boldsymbol{\Phi}$, or the i^{th} entry in the j^{th} basis function $\underset{\sim}{\boldsymbol{\phi}}_j$ (since the basis functions correspond to the columns of $\boldsymbol{\Phi}$), and η is the learning rate. This process occurs over *many* training image presentations. While the coefficients are re-initialised and optimised for each training image, the weights are optimised over a much longer time scale and require many training images to converge to an optimal solution.

Olshausen and Field trained their Sparsenet algorithm on 16x16 patches extracted from images of natural scenes. They experimented with a number of regularisation functions, such as $-e^{-x^2}$, $\log(1+x^2)$, and $|x|$, all of which gave qualitatively similar results [2, 6].

Their model was able to learn a complete set of localised, oriented, bandpass receptive

fields - qualitatively similar to those exhibited by simple cells in V1. Their landmark paper was the first to demonstrate the potential importance of sparseness in the visual cortex, and they concluded that maximising the sparsity of a code is sufficient to account for the response properties of V1 neurons [6].

1.3.2 Sparse-set Model

While the basis functions of Sparsenet looked qualitatively similar to those of neurons in V1, other work showed that they did not capture the full diversity of receptive fields in V1. To tackle this problem, Rehn and Sommer instead used the ℓ_0 norm to find sparse linear codes of natural images, finding solutions to:

$$\underset{\underset{\sim}{\boldsymbol{a}},\boldsymbol{\Phi}}{\operatorname{argmin}}\left(\left\|I-\boldsymbol{\Phi}\underset{\sim}{\boldsymbol{a}}\right\|_2^2+\lambda\|\underset{\sim}{\boldsymbol{a}}\|_0\right). \tag{1.25}$$

In practice this can only be solved approximately using algorithms such as matching pursuit [7]. In their paper, Rehn and Sommer made further approximations to the objective function that allowed the algorithm to be implemented as a Hopfield network [7], a type of recurrent neural network containing a single layer of nodes in which every node is connected to every other node. Feedback between nodes creates competition between nodes to represent an image, leading to sparsity. Their model was able to learn a much more diverse range of basis functions, learning both blob-like and unoriented structures, as well as the oriented bandpass edge-detector type basis functions that Sparsenet learns. They compared their model to the receptive fields of V1 neurons in a Macaque and found a much larger degree of similarity between the receptive fields of their model and of the Macaque [7].

1.3.3 Limitations

Sparse linear coding models such as those presented above are attractive for a number of reasons. First, they are designed to learn representations of natural images which can be directly compared to the receptive fields of biological neurons. It is also possible to make conclusions about the processes by which these models create sparseness, and whether or not the brain may be doing this. However, linear coding models like these cannot be extended to multiple layers. Nothing can be gained by stacking multiple linear coding models together, as this would just create another linear coding model for which a different but equivalent set of parameters can be learnt [3]. This is an issue as the visual system of the brain is hierarchical in nature.

While not strictly feed-forward, there is evidence of strong feed-forward connections from the LGN to V1, and from V1 to V2, and from V2 to further regions. Each layer in the hierarchy also may receive input from layers above (backwards connections) or from other parts of the brain, but it is apparent that a hierarchy exists wherein the neurons

higher up the hierarchy exhibit significantly more complicated response patterns than those below.

Neural networks, on the other hand, are capable of forming deep, multilayered structures. While they may be simpler in structure, and in some ways less biologically realistic than other forms of coding models, they offer the possibility of exploring how sparsity behaves across layers in a hierarchy. Furthermore, if sparse coding works well in the brain, it may also be a good strategy in neural networks, and this should be explored.

1.4 Deep Learning: Neural Networks

Neural networks have proven to be a simple yet powerful family of machine learning algorithms. Named for their resemblance to networks of neurons in the brain, neural networks consist of layers of artificial neurons, or nodes, connected by directed weighted edges. Information (such as the pixel values of an image) enters the network and moves between nodes via these weighted edges. Each node i has an *activation value*, a_i, which is the result of passing the inputs to the node, z_i through an activation function, $f(z_i)$. The way in which the connections between nodes in the network are arranged defines the network's *architecture*.

Neural networks generally begin with an *input layer*, which consists of the input data itself, and end in an *output layer* (the interpretation of this output depends on the function of the network). Any layers between the input and output layers are called *hidden layers*.

Networks with a single hidden layer are often called *shallow*, and networks with two or more hidden layers are called *deep*. Deep learning is the process of training deep networks to perform a wide range of tasks including classification, object detection, data dimensionality reduction, and denoising.

Numerous architectures have been imagined to tackle both supervised and unsupervised learning problems. Supervised models (which require labelled data for training) such as fully connected neural networks (FCNNs) perform well in classification and regression problems. Convolutional neural networks (CNNs) perform similarly well when the input data are images. Unsupervised models such as auto-encoders can be similar in structure to feed-forward networks, but instead are able to learn low-dimensional representations of data or to remove noise from data.

Here we introduce three common types of neural networks: Fully connected networks, convolutional networks, and auto-encoders.

1.4.1 Notation

- Let $\underset{\sim}{\boldsymbol{x}} = [x_1\ x_2 \cdots x_n] \in \mathbb{R}^n$ be a vector of input data points, where n is the dimension of the input. If the input is an $m \times l$ image, it will first be flattened into a vector of length $n = ml$, and the data will be said to be of dimension n .
- Let $\underset{\sim}{\boldsymbol{y}} = [y_1\ y_2 \cdots y_k] \in \mathbb{R}^k$ be the vector of outputs from the network, where k is the number of outputs. In a classification task, k will be equal to the number of classes.
- Let $\underset{\sim}{\boldsymbol{a}}^{(l)} \in \mathbb{R}^{N_l}$ be the vector of neuron activation values in the l^{th} hidden layer, where N_l is the number of nodes in that layer.
- let $\underset{\sim}{\boldsymbol{b}}^{(l)}$ be the vector of biases for the l^{th} layer. A bias is a constant term added to the input of a neuron before applying the activation function.
- Let $\mathbf{W}^{(l)}$ be the matrix of weights that lead into the l^{th} layer from the $(l-1)^{th}$ layer. The ij^{th} entry, W_{ij} (row i, column j), denotes the weighted edge connecting *to* the i^{th} node *from* the j^{th} node.
- Let $\underset{\sim}{\boldsymbol{w}}_i^{(l)}$ be the vector of weights connecting the nodes from the $(l-1)^{th}$ layer to the i^{th} node in the l^{th} layer. $\underset{\sim}{\boldsymbol{w}}_i^{(l)}$ corresponds to the rows of $\mathbf{W}^{(l)}$.
- Let $\underset{\sim}{\boldsymbol{z}}^l = \mathbf{W}^{(l)}\underset{\sim}{\boldsymbol{a}}^{(l-1)} + \underset{\sim}{\boldsymbol{b}}^{(l)}$. $\underset{\sim}{\boldsymbol{z}}^l$ is the vector of activation values in layer l *before* applying the activation function, f. z_i may be referred to as the internal state of node i.
- Let $f(z)$ denote the *activation function*. Some common activation functions are:
  - Sigmoid: $\sigma = \dfrac{1}{1+e^{-z}}$
  - Hyperbolic Tangent: $\tanh = \dfrac{e^z - e^{-z}}{e^z + e^{-z}}$
  - Rectified Linear Unit: ReLU $= \max(0, z)$

Additionally, let $L-1$ be the number of *hidden layers* in the network, excluding the input and output layers. With this labelling, the output layer is the L^{th} layer, and the input layer is the 0^{th} layer of the network.

1.4.2 Fully Connected Neural Networks

Fully Connected neural networks (FCNNs) (sometimes called "vanilla" neural networks or multilayer perceptrons) are the simplest type of neural network, and consist of organised layers of nodes with connections between neighbouring layers, but not within layers.

Nodes in one layer receive their inputs from nodes in the previous layer, and feed their outputs forward to nodes in the next layer. Fully connected networks are *fully connected*, so the internal state of one node is calculated as the weighted sum of *all* activation values in the previous layer. Information flows strictly forward from the input layer to the output layer.

FCNNs act as *universal approximators*: A multilayer fully connected network can approximate any continuous function to any degree of accuracy (if the activation function is non-polynomial) [32]. The ultimate goal of a feed-forward neural network then is to approximate some mapping $y = f(x;\theta)$ from a set of data x to a set of outputs, or predictions, y by learning the values of θ that best fit the function.

Fully connected networks are particularly useful in classification problems, where the value of each output represents the network's "confidence" that the input data belongs to a given category, or class. Regression problems can also be represented as neural networks.

Neural networks are best explained with an example. Consider the simplest case of a FCNN with a single output node and no hidden layers, as in Figure 1.3. This network takes three inputs, and gives a single valued prediction. The internal state of the output node is a weighted sum of the inputs, and the output is calculated by applying the activation function f, so that

$$y = f(\theta_1 x_1 + \theta_2 x_2 + \theta_3 x_3) \tag{1.26}$$

where θ_i are the parameters, or weights of the model, x_i are the inputs, or explanatory variables, f is the activation function and y is the outcome, or prediction of the model. The process of learning the weights, θ_i, in this network is equivalent to regression. If $f(\cdot)$ is the sigmoid function, the network is performing logistic regression, and if $f(\cdot)$ is a linear function the network is performing linear regression.

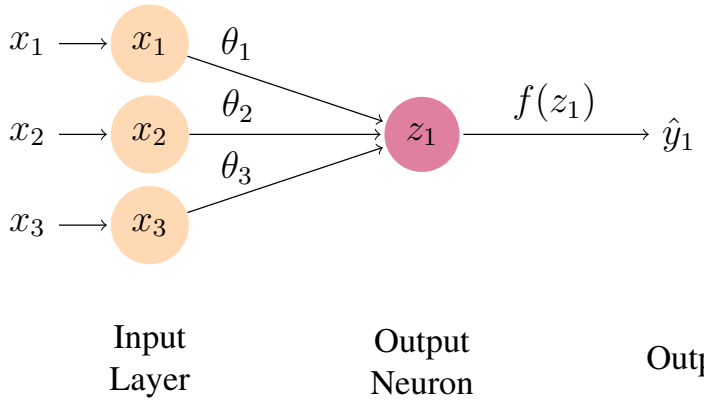


Figure 1.3: A single neuron network with three inputs. Regression can be seen as the special case of a single neuron network. With a sigmoid activation function on the output layer, the process of training this network is equivalent to sigmoid regression. (Created using code adapted from TikZBlog [33])

The utility in neural networks is that they can be easily expanded by either adding more layers, or more nodes to each layer. In regression, a non-linear decision boundary between explanatory variables may be found by adding higher-order terms to the model (e.g. $y = f(\theta_1 x_i^2 + \theta_2 x_2^2 + \ldots + \theta_n x_n^2)$). In networks, non-linear decision boundaries are formed by adding more layers or nodes to the network, in combination with using a non-linear activation function.

While the network in Figure 1.3 represents a relatively simple function (Equation 1.26), a deep network with multiple hidden layers represents the composition of a number of these simple functions:

$$f(x;\theta) = f^{(L)}(f^{(L-1)}(\ldots f^{(0)}(\underset{\sim}{\boldsymbol{x}}))) \tag{1.27}$$

Where L is the number of layers in the network

Figure 1.4 shows a network with three inputs, one hidden layer, and three outputs. There are a range of non-linear activation functions that can be used in hidden layers, including those presented in Section 1.4.1. However, the activation function for the *output layer* may differ depending on the task being performed.

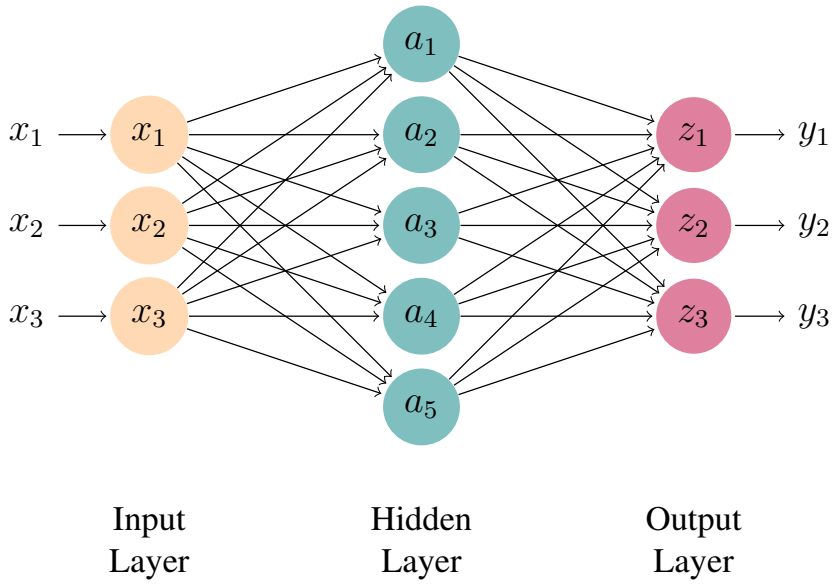


Figure 1.4: A fully connected neural network with a single hidden layer. (Created using code adapted from TikZBlog [33])

In binary classification problems, the output layer has a single node using the sigmoid activation function. Sigmoid gives an output between 0 and 1, and the output can be considered as the "confidence", or probability, that the input belongs to one classes (with the probability that the input belongs to the other class being $1 - y$).

The softmax function (Equation 1.28) is used as the activation function for the output layer in multiclass classification problems (when there are more than 2 output nodes).

This generalises the sigmoid to multiple outputs. It ensures that all the output values sum to 1, meaning each output can be interpreted as the probability that the input data belongs to that class. Sigmoid is equivalent to softmax with $K = 2$.

Definition 1.4.1. *(Softmax)* $\sigma : \mathbb{R}^K \mapsto [0,1]^K$

$$\sigma\left(\underset{\sim}{z}\right)_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \text{ for } i = 1, \ldots, K \text{ and } \underset{\sim}{z} \in \mathbb{R}^K \tag{1.28}$$

1.4.3 General Fully Connected Networks: Feed-forward Equations

The output of any feed-forward network is easily calculated using the feed-forward equations (Equation 1.30). In a fully connected feed-forward network, each node in layer l receives inputs from every node in the previous layer via the corresponding weighted connections, so that the activation value of node i in layer l is a linear combination of the activities in the previous layer, plus a bias term,

$$a_i^l = f(\underset{\sim}{w}_i^{(l)} \cdot \underset{\sim}{a}^{(l-1)} + b_i^{(l)}). \tag{1.29}$$

Using vector notation, where $\underset{\sim}{a}^{(l)}$ represents the activation of each node in layer l, the feedforward equation can be re-written as

$$\underset{\sim}{a}^{(l)} = f(\mathbf{W}^{(l)}\underset{\sim}{a}^{(l-1)} + \underset{\sim}{b}^{(l)}) = f(\underset{\sim}{z}^{(l)}). \tag{1.30}$$

Hence the i^{th} output of a network with L hidden layers is a composition of $L + 1$ functions

$$\begin{aligned} y_i &= \sigma(\underset{\sim}{z}^{(L)})_i \\ &= \sigma(\mathbf{W}^L\, f(\underset{\sim}{z}^{(L-1)}) + \underset{\sim}{b}^{(L)})_i \\ &= \sigma\left(\mathbf{W}^{(L)}\, f\big(\ldots f(\mathbf{W}^{(1)}\underset{\sim}{x} + \underset{\sim}{b}^{(1)})\big) + \underset{\sim}{b}^{(L)}\right)_i \end{aligned}$$

where $\underset{\sim}{x} = \underset{\sim}{a}^{(0)}$ is the input data. Note that in practice, inputs are usually *batched* together in an input matrix, $\mathbf{X}$, so that the calculations are performed on multiple data-points at once. The input is then a matrix in which each column represents an individual data-point. As a result, the activation values are also represented as matrices where each column gives the activation values resulting from an individual data-point. For example, in image classification the image is first flattened into a vector. The input $\mathbf{X}$ is then a matrix with rows corresponding to individual pixels in the image, and the columns correspond to separate images. Batching data is computationally efficient, but

makes no difference to the form of the feed-forward equations and so will be ignored for simplicity.

In classification tasks, the prediction of the network is the class corresponding to the output with the largest value

$$\hat{y} = \text{argmax}\, y_i$$

1.4.4 Learning: Backpropagation and Gradient Descent

We have seen that the output of the network can be easily calculated using the feed-forward equations, but in order to use a neural network as a model for prediction or classification, the correct weights and biases that best fit the data must be found. For simple linear regression the optimal set of weights and biases can be found analytically (although this is rarely done), but for non-linear regression this is not possible. Instead, the optimal set of weights and biases is found by performing gradient descent on an objective function, called the *loss function*.

The loss function is a function of both the network predictions, and the true labels of the data, and represents how closely the distribution of outputs from the network, given a set of inputs, resembles the true set of labels in the data. It plays the same role as the cost function used to learn linear codes. Fully connected networks are discriminative models, so a well trained network will closely model the conditional probability distribution, $p(y|x)$, of a set of outcomes or labels, y, given some explanatory data, x.

The most commonly used loss function for FCNNs is the *Cross Entropy Loss function*.

Definition 1.4.2. *(Cross Entropy)*

$$\mathscr{L} = CE(y,t) = -\sum_{i=1}^{k} t_i \log(y_i) = -\log \prod_{i=1}^{k} (y_i)^{t_i} \tag{1.31}$$

where t_i is the label - a binary variable indicating if the given data point belongs in class i ($t_i = 1$ if it does, and 0 if it does not), and y_i is the i^{th} output of the network.

The goal of learning is to alter the weights and biases so that the loss is minimised. Learning can be performed by presenting one image at a time to the network and calculating the loss (online learning), but generally the loss is averaged over a batch of images. In this way, the optimal weights are found by performing gradient descent on the loss function averaged over multiple images (this averaged loss function is often called the cost function),

$$J = \frac{1}{m} \sum_{n=1}^{m} CE(y^{(n)}, t^{(n)}) \tag{1.32}$$

where here m is the number of training images in each batch, and $y^{(n)}$ and $t^{(n)}$ refer to the output and label associated with image n.

Gradient descent is a simple process in which the parameters being learnt are iteratively changed by an amount proportional to the gradient of the cost function. The cost function can be imagined as a n-dimensional valley where the coordinates are given by the current parameters in the model. The goal is to find which coordinates represent the bottom of the valley, and these coordinates may be found by simply moving down the valley in the direction of steepest descent.

In each step of gradient descent, the weights are updated to be

$$\mathbf{W}_{new}^{(l)} = \mathbf{W}_{old}^{(l)} - \alpha \nabla_w \frac{1}{m} \sum_{n=1}^{m} CE(y^{(n)}, t^{(n)}), \tag{1.33}$$

where α is the *learning rate*, a tunable parameter which determines the relative size of each gradient step. The biases are also learnt in the same manner. When the number of images in each batch, m, is equal to the size of the training dataset, optimisation using Equation 1.33 is called *batched gradient descent*. When $m = 1$, it is called *Stochastic gradient descent* (SGD). Most often, a batch size somewhere between these two extremes is used, in which case the procedure is referred to as *Mini-batch gradient descent* [34]. Many gradient descent algorithms exist which build upon Equation 1.33 using various techniques to increase optimisation speed, with adaptive momentum estimation (Adam) being popular [35]. The details of these algorithms are outside the scope of this thesis.

Backpropagation Equations

The update rule for each learnable parameter in a network is found by application of the chain rule – a process called *backpropagation*. The backpropagation equations describe the way in which each parameter will change over the course of training.

The fully connected networks used in this thesis will in general use the ReLU activation function in the hidden layers, the softmax function on the output layer, and the Cross Entropy loss function. For such fully connected networks with L layers, the update rules for the weights and biases in layers $l < L$ are

$$\Delta W_{ij}^{(l)} \propto \frac{\partial \mathscr{L}}{\partial W_{ij}^{(l)}} = \begin{cases} \sum_{n}^{N_L} \frac{\partial z_n^{(L)}}{\partial a_i^{(l)}} (y_n - t_n) a_j^{(l-1)}, \text{ if } z_i^{(l)} > 0 \\ 0 \text{ otherwise} \end{cases} \tag{1.34}$$

$$\Delta b_i \propto \frac{\partial \mathscr{L}}{\partial b_i^{(l)}} = \begin{cases} \sum_{n}^{N_L} \frac{\partial z_n^{(L)}}{\partial a_i^{(l)}} (y_n - t_n), \text{ if } z_i^{(l)} > 0 \\ 0 \text{ Otherwise} \end{cases} \tag{1.35}$$

where $\dfrac{\partial z_i^{(L)}}{\partial a_i^{(l)}}$ can be evaluated using the chain rule, and N_L is the number of nodes in the output layer. For parameters in layer $l = L$, the update rules are

$$\Delta W_{ij}^{(L)} \propto \frac{\partial \mathscr{L}}{\partial W_{ij}^{(L)}} = (y_i - t_i) a_j^{(L-1)} \tag{1.36}$$

$$\Delta b_i \propto \frac{\partial \mathscr{L}}{\partial b_i^{(L)}} = (y_i - t_i) \tag{1.37}$$

The backpropagation equations for networks using different cost functions or activation functions will differ slightly. See Appendix B.1 for the derivation of these equations.

1.4.5 Convolutional Neural Networks

Convolutional neural networks (CNNs) are similar to feed-forward networks - they contain organised layers of nodes, with connections between layers but not within, and information flows strictly forward through the network. However they differ in the operations performed in each layer. Rather than the activation of each node in a layer being a linear combination of all activations in the previous layer, they are instead calculated as the *convolution* of the previous layer with a kernel. Convolutional networks excel at problems involving image data, such as image classification, object localisation, and object detection [36].

The convolution of a discrete signal, x, with a kernel h is defined as

Definition 1.4.3. *(Convolution)*

$$x * h = \sum_{\tau=-\infty}^{\infty} x(\tau) h(t - \tau). \tag{1.38}$$

The convolution is closely related to the cross-correlation, in which the kernel is not reversed in τ:

Definition 1.4.4. *(Cross-correlation)*

$$x \tilde{*} h = \sum_{\tau=-\infty}^{\infty} x(\tau) h(\tau - t). \tag{1.39}$$

Despite the name, it is usually the cross-correlation that is used in CNNs. Convolutions are often desirable as they are commutative - the order in which a series of convolutions are applied does not change the end result. However, commutativity is not necessary in neural networks, as non-linear activation functions are usually applied immediately after a convolution, so convolutions are never chained together. Additionally, the kernel h is arbitrary as it is a set of learnt parameters, and as such the cross-correlation kernel $h(\tau - t)$ may be learnt as an equivalent convolution kernel $h'(t - \tau)$ – there is no real distinction.

Whereas previously it has been easiest to visualise nodes in rows and place their activations in vectors, in this case it is easier to imagine the activations of nodes in a layer as pixels of an image. These arrays of activations are sometimes called *feature maps*.

Let $a_{i,j}$ be the pixel in row i column j of the image. Let $\mathbf{H}$ be the kernel, with elements $H_{l,m}$ corresponding to the entry in row l column m. The kernel can be thought of as a filter and is analogous to the weights of a feed-forward network. It is a small array of weights (e.g., 3×3, 5×5, 7×7), that "scans" its way along the input image. At each step, the filter is convolved with the set of pixels it overlaps, and the output of this operation becomes a single pixel in the output image. Figure 1.5 shows this operation.

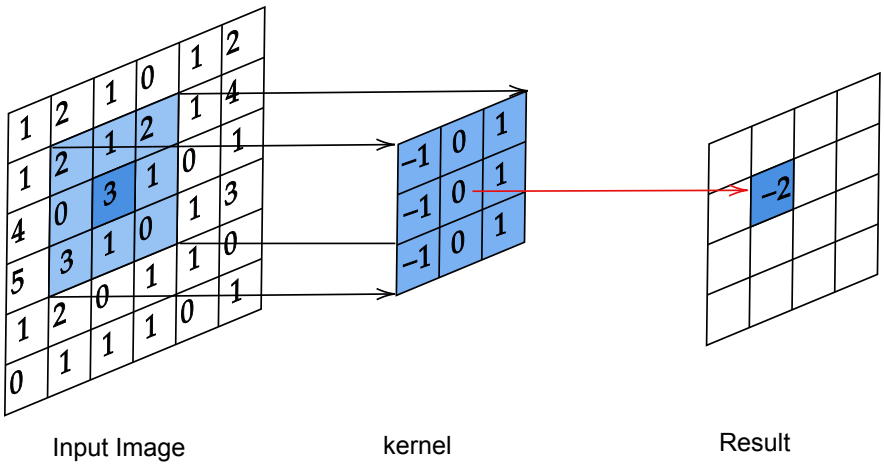


Figure 1.5: Convolution of an image by a 3×3 kernel. This particular kernel is a stereotypical edge detector. The input image is 6×6 in size, with no padding. Hence with a stride length of 1, the output image 4×4 in size.

The purpose of the kernel is to learn "features" of the data. CNNs are typically robust to translations of features in an image, since a kernel (having learnt a particular feature) is convolved with the entire image.

The activation value of the $(i, j)^{th}$ output node is given by [37]:

$$a_{i,j}^{(l+1)} = \sum_{l=0}^{n_k} \sum_{m=0}^{n_k} H_{l,m}^{(l)} a_{i+l,j+m}^{(l)} \tag{1.40}$$

Where the superscript l again refers to the layer in which the node belongs, and n_k is the kernel width (assuming a square kernel). The size of the output of this operation is

$$\text{Output size} = \frac{n_x + 2P - n_k}{S} + 1, \tag{1.41}$$

where n_x is the side length of the input, n_k is the kernel size, P is the padding which indicates how much zero padding should be placed around the input image, and S is the stride length, which indicates how many pixels the kernel should translate by with each stride [36].

In order to learn multiple features of a dataset, a CNN will usually use multiple kernels per layer, producing a separate output "image" for each kernel. For example, if the first convolutional layer uses four kernels, the next layer will consist of four sets of nodes. Each set of nodes is called a channel.

When a convolution is performed on multiple channels (whether it be an RGB image with 3 channels, or a layer in which multiple kernels were used), the individual channels are first "stacked" to form a 3D array of values. A "full-depth" 3-dimensional convolution is then performed on all channels at once – the width and height of the kernel are chosen as before, however the *depth* of the kernel is equal to the number of channels the convolution is being performed on, and the *same* kernel is used for each channel (i.e. the kernel is simply copied in the depth direction). This way, if 16 kernels are convolved with a 6 channel layer, there will be 16 output channels.

Figure 1.6 shows the structure of a CNN which takes in 28×28 greyscale images. In the first layer, six 7×7 kernels are convolved with the input, producing 6 output channels. In the second layer, 16 kernels are convolved with the full depth of channels in the previous layer.

Convolutional networks typically have a small fully connected network attached to the end which acts as a classifier. The convolutional layers act as a feature extractor, learning important structures in the input data, allowing the final fully connected layers to classify the input based on these features. To transition into the fully connected layers, the final set of feature maps are *flattened* into a vector of activations which act as the inputs to the fully connected layers.

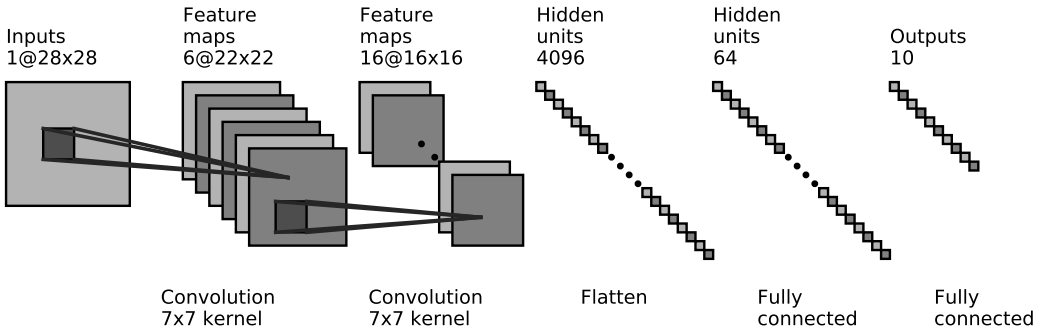


Figure 1.6: A convolutional neural network with two convolutional layers. This network takes 28×28 greyscale images as an input, and places them in one of ten classes. It uses 7×7 kernels, with six kernels used in the first layer and 16 used in the second layer. This exact architecture is used later in Chapter 4 for the two layer convolutional networks. (This figure was created by adapting code by Ding, 2018 [38].)

1.4.6 Autoencoders

Following the same patterns as the previous types of networks, autoencoders are also architecturally similar to feed-forward networks. Autoencoders presented in this thesis are fully-connected and strictly feed forward, with connections between neighbouring layers but not within layers. Unlike previous architectures however, their goal is not classification and regression. Rather, autoencoders are typically used to find low-dimensional representations of data. Autoencoders employ a "bottleneck" architecture - each successive layer after the input decreases in size, until a central "encoding" layer. After this layer, the architecture is an exact copy of the first half of the network in reverse. The first half of the network is called the "encoder", while the second half is called the "decoder" - Figure 1.7 shows an example of this. Trained together, these two components aim to learn an identity function $f(\underset{\sim}{x}) = \underset{\sim}{x}$. This generally would be useless, however because the central encoding layer is of a smaller size than the input data, the network is forced to find a low-dimensional representation of the input. The activities of that layer can then be extracted and used. Autoencoders are useful in signal compression and de-noising.

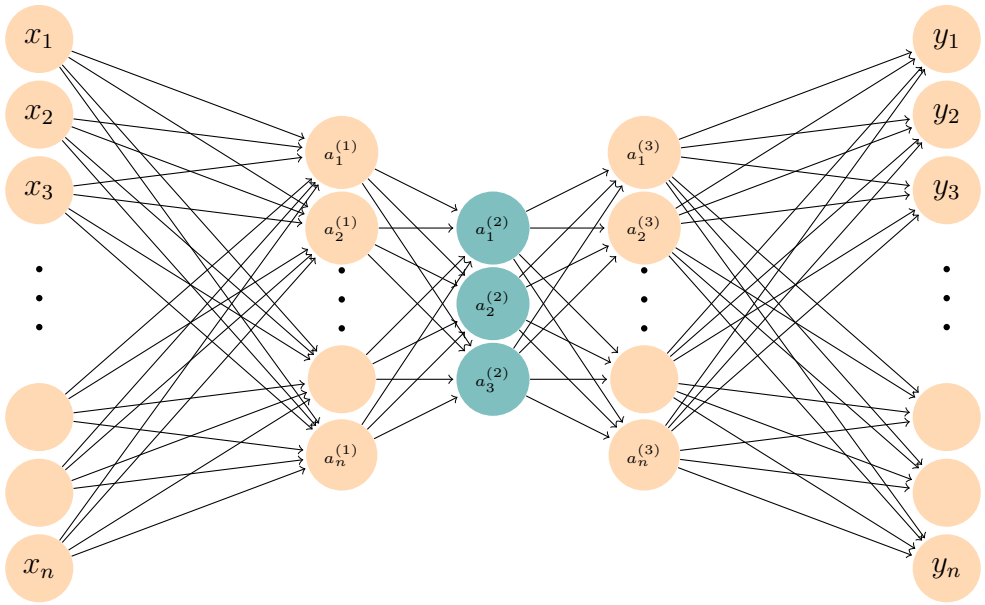


Figure 1.7: An autoencoder employing the stereotypical 'bottleneck" architecture. This autoencoder accepts n inputs, and has three hidden layers. The central layer can be called the "coding" layer, or latent space. The component of the autoencoder to the left of the coding layer is called the *encoder*, and the component to the right is called the *decoder*.

In order to learn a faithful reconstruction of the input data, autoencoders are trained using the Mean Square Error cost function rather than Cross Entropy.

$$MSE = \frac{1}{m}\sum_{i=1}^{m}\|\underset{\sim}{\boldsymbol{x}} - \underset{\sim}{\boldsymbol{y}}\|_2^2 \tag{1.42}$$

Where $\|\cdot\|_2$ is the ℓ_2 norm and m is the number of training images.

CHAPTER 2

Sparsifying Neural Networks

Chapter 1 discussed how sparse linear coding models may be formed using models such as Sparsenet, and how forming such linear codes leads to a set of learnt basis functions that resemble the receptive fields of neurons in V1. In order to examine how sparsity behaves in neural networks, we need a method of penalising the network for having too much activity.

2.1 Regularisation

It is common practice to add additional terms to the cost function of a neural network in order to regularise the parameters of the network. Often, the weights of the model are regularised in order to reduce their size and ensure over-fitting does not occur. This is done by adding a regularisation term, S, consisting of either the ℓ_1 norm of all weights in the network, or the squared ℓ_2 norm of the weights, as shown below:

$$S = \|\mathbf{W}\|_1 = \sum_{ij} |W_{ij}|$$

$$S = \|\mathbf{W}\|_2^2 = \sum_{ij} (W_{ij})^2$$

As with the linear coding models described previously, regularisation adds an extra penalty to the cost function being optimized. This is analogous to LASSO regression, in which the ℓ_1 norm of the parameters is used, and RIDGE regression in which the L2 norm is used.

In this thesis we wish to use a regularisation term that will encourage sparsity in the activation values rather than in the weights. Louizos et al [39] presented a method for ℓ_0 norm regularisation of the weights in neural networks. As with the "hard sparseness" form of Sparsenet, the ℓ_0 norm is challenging to work with due to its non-differentiability. By extending this method so that it acts on a set of weights at once (rather than on individual weights), it is possible to regularise all weights leading into a single neuron

together. For example, if the weight of one connection leading into a neuron became zero due to the regularisation, then all other weights leading into that neuron would also go to zero. Once training is complete, the neurons for which incoming weights have gone to zero will essentially be "pruned" from the model. This may be useful as a method for optimising the architecture of a network, but neuron pruning is not equivalent to enforcing sparse activations, and so is not useful here.

While the ℓ_0 norm appeared important in the linear coding models presented in Chapter 1, in this thesis the ℓ_1 norm is instead chosen the difficulties associated with the non-differentiability ℓ_0 norm. Here, the activity of networks is regularised using the ℓ_1 norm of the *activations* in the network, so that the regularisation term S for a given input is

$$S(\underset{\sim}{\boldsymbol{a}}) = \sum_{l=1}^{L-1} \|\underset{\sim}{\boldsymbol{a}}\|_1 = \sum_{l=1}^{L-1} \sum_{i}^{N_l} |a_i^{(l)}|. \tag{2.1}$$

The ℓ_1 penalty for a neural network is here defined as the sum of the absolute value of the activation values for all nodes in all hidden layers (input and output layers are not regularised). It then follows that the loss associated with a given input for classification tasks is

$$\begin{aligned} \mathscr{L}(\underset{\sim}{\boldsymbol{y}}, \underset{\sim}{\boldsymbol{t}}, \underset{\sim}{\boldsymbol{a}}) &= CE(\underset{\sim}{\boldsymbol{y}}, \underset{\sim}{\boldsymbol{t}}) + \lambda S(\underset{\sim}{\boldsymbol{a}}) \\ &= -\sum_{i=1}^{k} t_i \log(y_i) + \lambda \sum_{l=1}^{L-1} \sum_{i}^{N_l} |a_i^{(l)}| \end{aligned} \tag{2.2}$$

and over a batch of images, the cost function is

$$J = \frac{1}{m} \sum_{n=1}^{m} \mathscr{L}(\underset{\sim}{\boldsymbol{y}}^{(n)}, \underset{\sim}{\boldsymbol{t}}^{(n)}, \underset{\sim}{\boldsymbol{a}}^{(n)}) \tag{2.3}$$

where $\underset{\sim}{\boldsymbol{y}}^{(n)}, \underset{\sim}{\boldsymbol{t}}^{(n)}$, and $\underset{\sim}{\boldsymbol{a}}^{(n)}$ are the output, label, and activations associated with the n^{th} training image, and m is the number of training images. The coefficient λ is the regularisation coefficient, which scales the trade off between sparsity and accuracy – a larger λ value will lead to greater sparsity, but likely at the cost of performance. In theory this could be done with any continuous regularisation function that associates a cost with the activity of each neuron. This cost function was used to train all fully connected networks and CNNs presented in this thesis.

Similarly, autoencoders were trained using the reconstruction mean square error as follows:

$$\begin{aligned} \mathscr{L}(\underset{\sim}{\boldsymbol{y}}, \underset{\sim}{\boldsymbol{x}}, \underset{\sim}{\boldsymbol{a}}) &= MSE(\underset{\sim}{\boldsymbol{y}}, \underset{\sim}{\boldsymbol{x}}) + \lambda S(\underset{\sim}{\boldsymbol{a}}) \\ &= \|\underset{\sim}{\boldsymbol{x}} - \underset{\sim}{\boldsymbol{y}}\|_2^2 + \lambda \sum_{l=1}^{L-1} \sum_{i}^{N_l} |a_i^{(l)}| \end{aligned} \tag{2.4}$$

The alternative of regularising with the ℓ_0 norm suggested by Rehn and Sommer would not be easily implemented in a deep network framework in which gradient descent is used as the optimisation procedure.

While activity regularisation in neural networks is uncommon, it isn't new. The ReLU activation function in combination with an ℓ_1 activation penalty has been shown to promote sparseness in deep autoencoders [40]. It is likely that the ReLU activation function is even essential in creating truly sparse networks, as functions such as the sigmoid function do not allow the activation of neurons to reach true zero.

2.2 General Implementation

In order to explore the role of sparsity in deep neural networks, a number of network architectures were trained using the ℓ_1 penalty in Equation 2.1. Specifically, feed-forward networks, convolutional networks, and autoencoders of various sizes were tested. The exact architectures used in each case will be presented in their respective sections.

In each case, networks were trained with a range of values for the regularisation coefficient, λ. This is necessary as λ is an arbitrary constant that would need to be tuned if activity regularisation were implemented in practice. The specific range of λ values used in each case was chosen as a result of preliminary testing so that values were not so large as to prevent training, or so small that the effect of regularisation was negligible. For each combination of model size and λ value, 20 individual networks were trained. This was done because, although learning is a deterministic process, changes in initial parameters and the order in which training images are presented mean that separate networks will learn differently.

2.2.1 Training

All networks were trained using the Adam (adaptive moment estimation) optimization algorithm [35]. Adam adapts the learning rate for each parameter in the model individually to prevent any parameters from updating slowly (which occurs if their gradients are small), and uses momentum to increase the learning rate throughout the training process. Adam was chosen as it is easy to configure and allows much faster training times than traditional algorithms (time being a significant constraint). The details of this algorithm are outside the scope of this thesis.

Training data was presented to the network in batches of 100 images. The average loss over an entire batch was calculated, and the weights were updated once using the gradients calculated from this loss. An epoch passes once all batches have been presented to the network. To save time, each network was trained for only 15 epochs.

2.2.2 Performance Measures

For networks trained on datasets that are known to be balanced and well-behaved such as the MNIST dataset, accuracy was used to measure the network performance. Accuracy is simply defined as the proportion of correct predictions made by the network over an entire dataset;

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} = \frac{\text{Number of Correct Predictions}}{\text{Total Number of images}} \tag{2.5}$$

where $TP =$ number of true positives, $TN =$ true negatives, $FP =$ false positives, and $FN =$ false negatives.

For each network the median accuracy was calculated from the 20 repetitions of each test case, and the interquartile range (IQR) was reported, as well as maximum and minimum accuracies. This was done as the IQR and maximum and minimum accuracies gave a much more representative description of the spread of accuracies.

Both ℓ_1 and ℓ_0 measurements of network activity were recorded when testing the network. For each, the average activity *per image* of each network was calculated. This was then averaged over the 20 models trained for each test condition. The ℓ_0 activity for the entire network is given as follows:

$$\ell_0 \text{ activity} = \frac{1}{20}\sum_{r=1}^{20}\left[\frac{1}{m}\sum_{n=1}^{m}\sum_{l=1}^{L-1}\|\underset{\sim}{\boldsymbol{a}}^{(n)}\|_0\right]_r. \tag{2.6}$$

Similarly the ℓ_1 activity for the entire network is

$$\ell_1 \text{ activity} = \frac{1}{20}\sum_{r=1}^{20}\left[\frac{1}{m}\sum_{n=1}^{m}\sum_{l=1}^{L-1}\|\underset{\sim}{\boldsymbol{a}}^{(n)}\|_1\right]_r \tag{2.7}$$

where m is the number of test images, $\underset{\sim}{\boldsymbol{a}}^{(n)}$ are the activation values associated with the n^{th} test image, and r denotes which of the 20 networks for each test case is being tested. The Gini Index, defined in Equation 1.18 was also used to measure sparsity in some networks.

2.2.3 Software

Networks were implemented in *Python 3.8.7* using *PyTorch* 1.7.1 [8], a Python deep learning library.

The method for regularising a network by the ℓ_1 norm of the network's activation values was implemented into the training function (Appendix C.2). See Appendix C for the training function, testing functions, and example implementations.

2.2.4 Datasets

Two datasets were used to train the networks. All networks were trained and tested on the MNIST dataset of handwritten digits [41]. MNIST consists of $70,000$ 28×28 greyscale images of handwritten digits training images. There are $60,000$ training images, which were split into $50,000$ training and $10,000$ validation images, and $10,000$ test images.

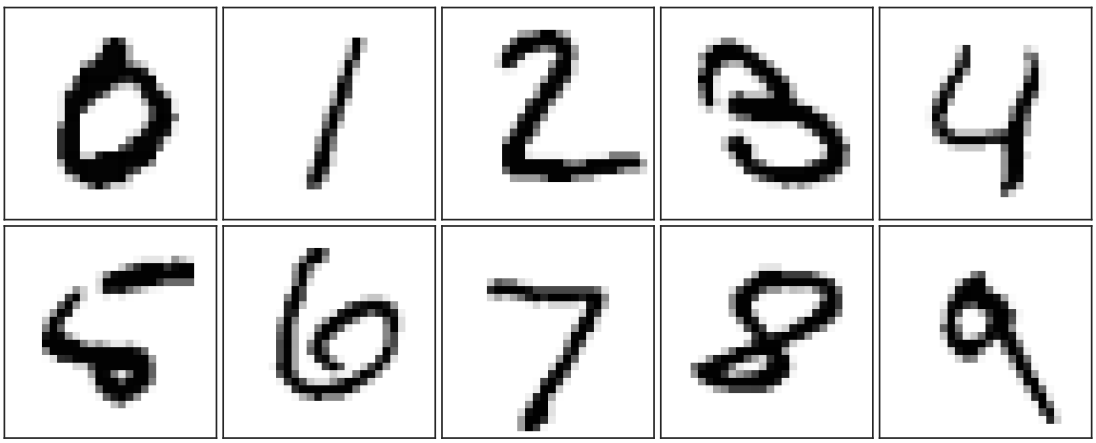


Figure 2.1: Examples of the ten classes present in the MNIST database of handwritten digits.

Convolutional networks were additionally tested on CIFAR-10 [42]. CIFAR-10 consists of $60,000$ 32×32 RGB images containing 10 classes of image, shown in Figure 2.2.

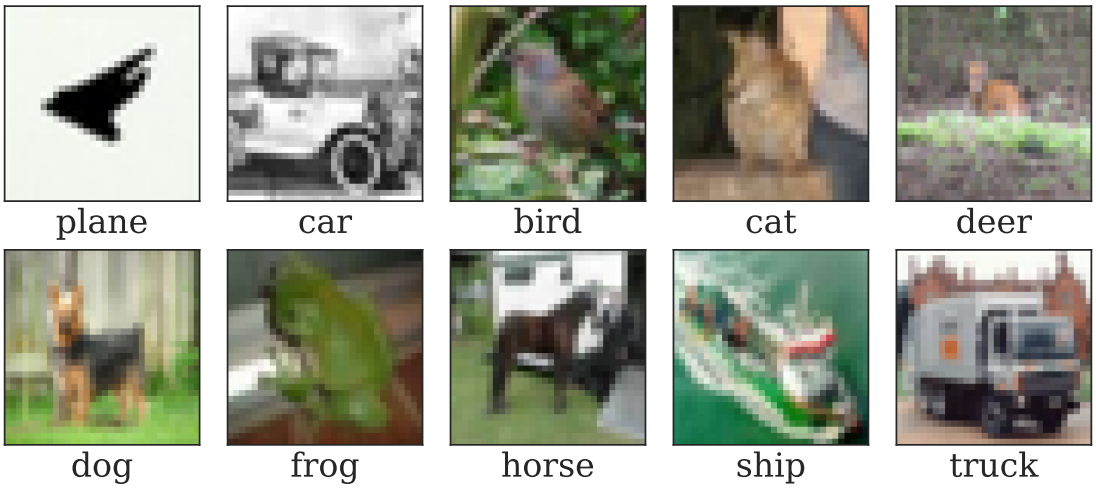


Figure 2.2: Examples of the ten classes present in the CIFAR-10 dataset.

CHAPTER 3

Fully Connected Networks

This chapter presents the results of training fully connected networks on the MNIST dataset while enforcing sparse activations. We begin by discussing networks with a single hidden layer, and compare how sparsity changes with the use of sigmoid and ReLU activation functions. We then discuss networks with two and three hidden layers in order to see how sparsity might change across the layers in a hierarchical structure. The main goals of this chapter are:

- To determine how effective is the method of activity regularisation used in this thesis at minimising the number of active nodes in a network, and how this changes the dynamics of the network.
- To determine whether sparsely activated networks have an advantage over conventional networks, either in performance or some other area.
- To explore how activity changes across various layers of a deep network when regularisation is applied to the entire network.

In each case, networks with 8, 32, 64, 128, and 256 nodes in each hidden layer were trained. Networks with multiple hidden layers had the same number of nodes in each layer. In this chapter, networks in which the ReLU activation function was used will be referred to as ReLU networks, and networks which employed the sigmoid activation function will be referred to as sigmoid networks.

The regularisation coefficient, λ, was varied from 0 to 0.12 in steps of 0.008, unless otherwise specified. For each combination of layer size, network depth, and λ, 20 networks were trained and tested.

3.1 Single Layer Networks

3.1.1 Fully Connected Networks are Overactive

Figure 3.1 shows the median accuracy of ReLU networks with a single hidden layer trained and tested on the MNIST dataset and regularised to varying degrees (by varying λ). For small values of λ there was little to no drop in accuracy when compared to unregularised networks. As λ was increased, accuracy slowly dropped until a critical λ value was reached, at which point accuracy decreased dramatically.

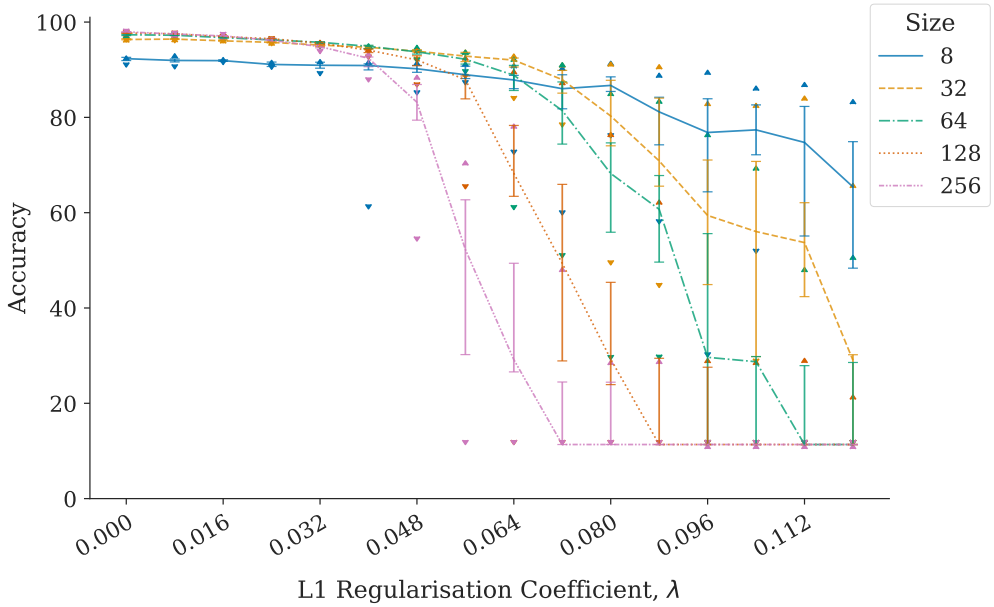


Figure 3.1: Median accuracy of five sizes of fully connected neural networks with a single hidden layer, for different values of λ. Accuracy is defined as the proportion of correct predictions made by the network out of a set of test images. Error bars show the inter-quartile range, upwards pointing triangles show the maximum accuracy, and downwards pointing triangles show the minimum accuracy.

Figure 3.2 shows the average number of active nodes per image (i.e., the number of nodes that activate on average for a single input) in the hidden layer for each value of λ. Unregularised networks had a large proportion of hidden neurons active for any given image presentation. However, only a small λ value was required to cause a massive reduction in the number of active nodes per image. If λ was too large, the network was encouraged to have zero nodes active on average, in which case the network was entirely inactive for nearly all inputs. When this occurs the only inputs into the output layer are the biases associated with the output layer, and as a result the network makes the same prediction regardless of the input. Note also the increase in the spread of values for network accuracy for larger values of λ. This increase occurred around the “critical

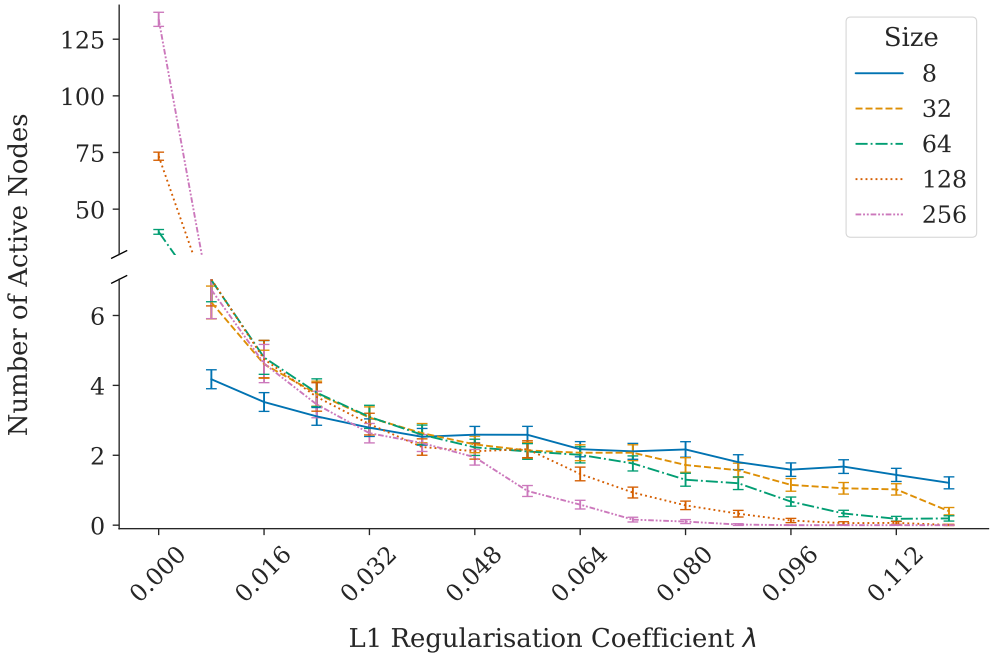


Figure 3.2: Average number of active nodes per input image in the hidden layer of single layer fully connected networks. Values were calculated by measuring the average number of nodes active per image for each network, and averaging over the 20 models trained for each test condition. Error bars indicate standard deviation. Note also the broken y-axis.

value" of λ at which accuracy began to decrease significantly. Past this point training became much more unstable. Often for large λ values training would stagnate, with the accuracy of the network reaching around 11% and improving no further. This was the best the network could do if it made the same prediction for every input (in this case the network is predicting that every input is the digit 1). The minimum accuracy of 11% simply indicates that the digit 1 comprised 11% of the test dataset, and was the class containing the most images.

Comparing Figure 3.1 and 3.2, one can clearly see the trade-off between network sparsity and accuracy. Small values of λ resulted in a significant decrease in the number of active nodes per image with only a marginal decrease in accuracy. Even with a small regularisation coefficient of $\lambda = 0.008$, the average activity of each network dropped to below 8 nodes for a given input, regardless of the network size. Larger values of λ continued decreasing the average number of active nodes, but to a much lesser degree and with a much greater impact on network accuracy.

While the regularised networks had fewer than 8 active nodes per input, this does not mean that only 8 nodes are needed to classify digits. Instead it is a different set of nodes that are active for each image, and the size of that set is on average less than 8. Because of this the larger networks are still far more accurate than the 8 node network, despite

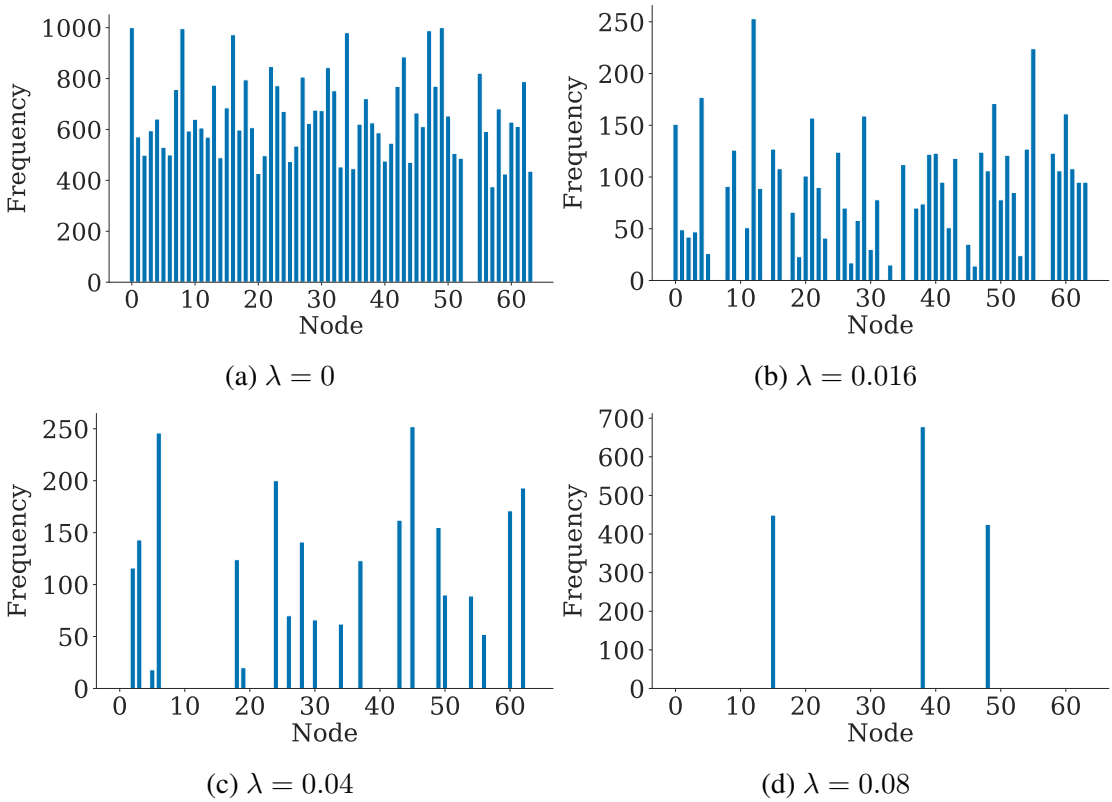
ultimately having similar levels of activity. This suggests that adding more nodes to the network allows it to have a greater representational capacity, but limiting network activity means that the network should learn the "simplest" model it can. Recall that the sparse coding models discussed in Chapter 1 were generally overcomplete. This was both because the visual system is thought to be overcomplete [7] and because an overcomplete code has greater flexibility in adapting to the structure of the input data [6]. In this case, there appears to be a similar advantage to having more neurons in the network than may be "necessary".

Recall that fully-connected networks approximate non-linear functions by learning the set of parameters that best fit the data. One challenge when using neural networks is the problem of model selection - deciding which architecture (i.e., number of layers and nodes, choice of activation function, etc.) will perform the best. An over-complicated network will be prone to over-fitting, just as an over-complicated model would be when performing regression. In a network that uses the sigmoid activation function (discussed in the next section) all nodes are active at all times, so the possibility of overfitting is much more likely if an over-complicated architecture is chosen. This problem is usually combatted in part by regularising the weights of the network. In this way some weights are encouraged to be small, decreasing the influence of their associated nodes.

Activity regularisation could in theory be an alternative to this when used in combination with the ReLU activation function. Instead of decreasing the effect of certain nodes by shrinking their associated weights, the weights are learnt in such a way that some nodes are completely inactive at least some of the time. During testing there was no obvious sign that activity regularisation decreased overfitting, however this was not thoroughly investigated and requires further work to make a conclusion. Activity regularisation has also previously been used to prevent numerical problems associated with the activation value of ReLU nodes exploding in size, since ReLU is unbounded for $x > 0$ [40].

Figure 3.3 shows the number of times each node in a 64 node network activated over the presentation of 1000 images, demonstrating two effects that activity regularisation had on the activity of ReLU nodes. First, as λ increased, the number of images for which each node activated decreased. This suggests that each node is becoming more selective, with a smaller set of data that will lead to its activation. At $\lambda = 0.08$, however the active nodes again began to respond to a greater number of inputs. This is because there were only 3 nodes that ever activated, so each node must activate for a greater range of inputs.

Second, regularisation increased the number of "dead" nodes. ReLU networks suffer from the "dying ReLU" problem, where some nodes will become entirely inactive and output zero for all inputs. This is typically a problem in deep networks [43]. Despite the network used in Figure 3.3 being a shallow network, even for $\lambda = 0$ there were two nodes that were inactive for all of the 1000 images presented. The number of dead nodes increased as λ increased until only 3 nodes survived for $\lambda = 0.08$, and eventually for large enough values of λ all neurons died.



(a) $\lambda = 0$

(b) $\lambda = 0.016$

(c) $\lambda = 0.04$

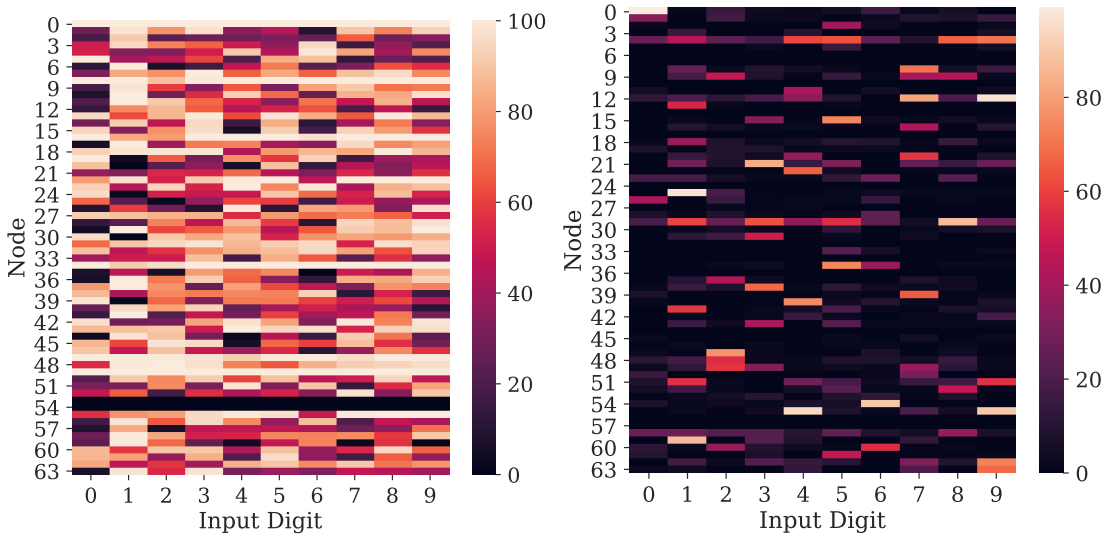
(d) $\lambda = 0.08$

Figure 3.3: The number of times each node in a 64 node network activated over the presentation of 1000 images for four values of λ. Some nodes appear to be "dead", and output zero for all inputs. Increasing λ has two effects here: increasing the number of dead nodes, and decreasing the total number of inputs each node activates for (unless λ is so large that all the work is handled by only a few nodes as in Subfigure (d)).

On one hand, dead nodes can be a problem as they prevent backpropagation of gradients along that path. However other experimental work suggests that dead ReLU nodes do not hurt gradient based learning so long as the gradient can propagate along *some* paths, i.e., so long as some nodes in a hidden layer have a non-zero activation value [40]. Given this then, it might be possible that dead nodes can aid in model selection - if a network has a large amount of dead nodes but is performing well, then the number of nodes in the network can clearly be reduced.

It is also interesting to note that networks with 64 nodes or more in the hidden layer had nearly identical accuracies, while the 32 node network had a slightly lower accuracy. These networks could be regularised to the point of having $4-5$ active nodes per image without a significant hit to accuracy. An average ℓ_0 activity of $4-5$ suggests that perhaps only 4 or 5 features are needed at a time to distinguish one digit from another. Figure 3.4 shows four heatmaps indicating the number of times each node in a 64 node network activated for each digit for four values of λ. When $\lambda = 0$, nearly all nodes activated at least once for every digit (each digit elicited activity from $60 - 62$ out of 64 nodes at

least once). In comparison, when $\lambda = 0.04$, each digit only elicited activity at least once from $13-16$ out of 64 nodes. This suggests that only $13-16$ nodes are needed per class of input, of which only 4 or 5 may be needed at any one time, and many of which can also be used in the recognition of other digits (hence the total number of nodes needed per node may be less than 13). As a consequence it is possible for a network to learn more efficient representations of the data than it otherwise might when unregularised.



(a) $\lambda = 0$
Accuracy = 97.34%

(b) $\lambda = 0.016$
Accuracy = 96.81%

(c) $\lambda = 0.04$
Accuracy = 95.11%

(d) $\lambda = 0.08$
Accuracy = 76.96%

Figure 3.4: Heatmaps showing the number of times each node in a 64 node network activated for each digit for different values of λ. Each 100 images from each class was presented, for a total of 1000 images.

3.1.2 Larger Networks are More Sensitive to Regularisation

Figure 3.1 also shows that the sensitivity of a network to the method of activity regularisation used in this thesis is positively correlated to the number of nodes in the network. Sensitivity to regularisation refers to the size of the critical λ value that results in a catastrophic drop in accuracy, where a smaller critical λ implies a greater sensitivity.

While the same λ values were tested for all fully connected networks, it is perfectly reasonable to use smaller λ values for larger networks. λ is simply a proportionality constant - it is the level of sparsity induced by the regularisation term that is important. One (possibly naïve) alternative to account for network size was to instead regularise by the activity *per node*, rather than the overall activity of the network, as follows:

$$C(y_i, t_i) = -\sum_{i=1}^{K} t_i \log(y_i) + \frac{\lambda}{n} \sum_{l=1}^{L-1} \sum_{i} |a_i^{(l)}|, \tag{3.1}$$

where n is the number of nodes in the network. This, however, simply resulted in the smaller networks being penalised more harshly instead, and so was not explored any further.

Figure 3.5 shows the average ℓ_1 activity per image in the same networks. As expected, the total ℓ_1 activity over the network decreased as λ increased, showing that the ℓ_1 regularisation is in fact working to decrease the ℓ_1 activity, as well as ℓ_0 activity. However, the largest networks tended to have the lowest level of ℓ_1 activity, while the 8 node network had the largest. This was not the case for unregularised networks, where ℓ_1 activity increased with network size (Table 3.1).

Table 3.1: ℓ_1 activity for different sizes of unregularised single layer fully connected networks ($\lambda = 0$) and their standard deviation. Unsurprisingly, the largest networks have the most activity, while the smallest networks have the least.

Size	ℓ_1 Activity for $\lambda = 0$
8	46.08 $\pm$ 3.53
32	78.23 $\pm$ 3.71
64	113.49 $\pm$ 5.75
128	157.77 $\pm$ 7.54
256	220.54 $\pm$9.69

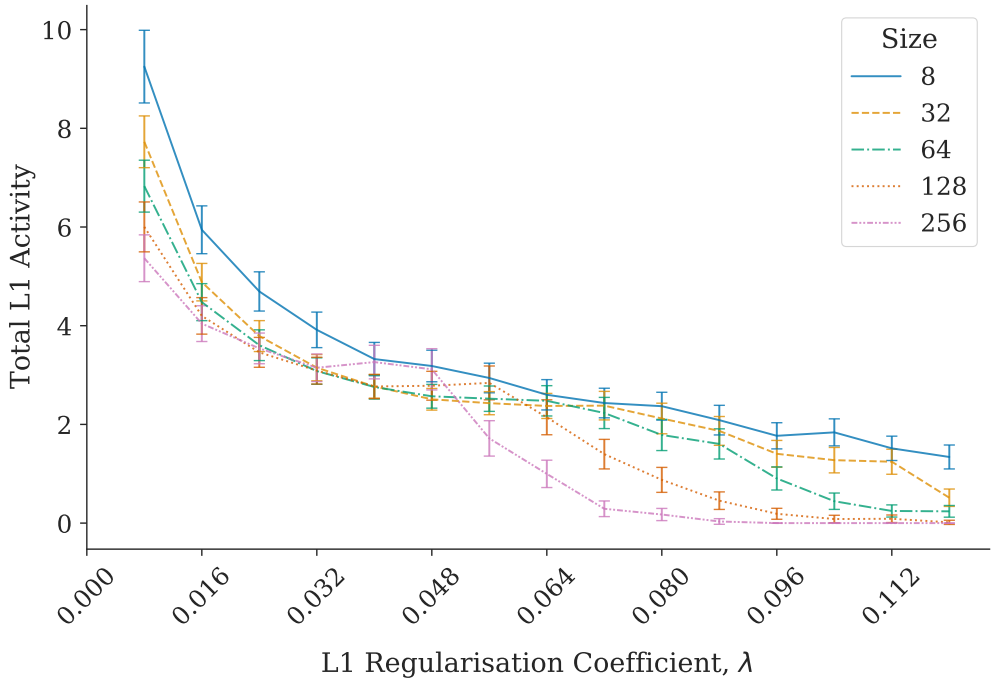
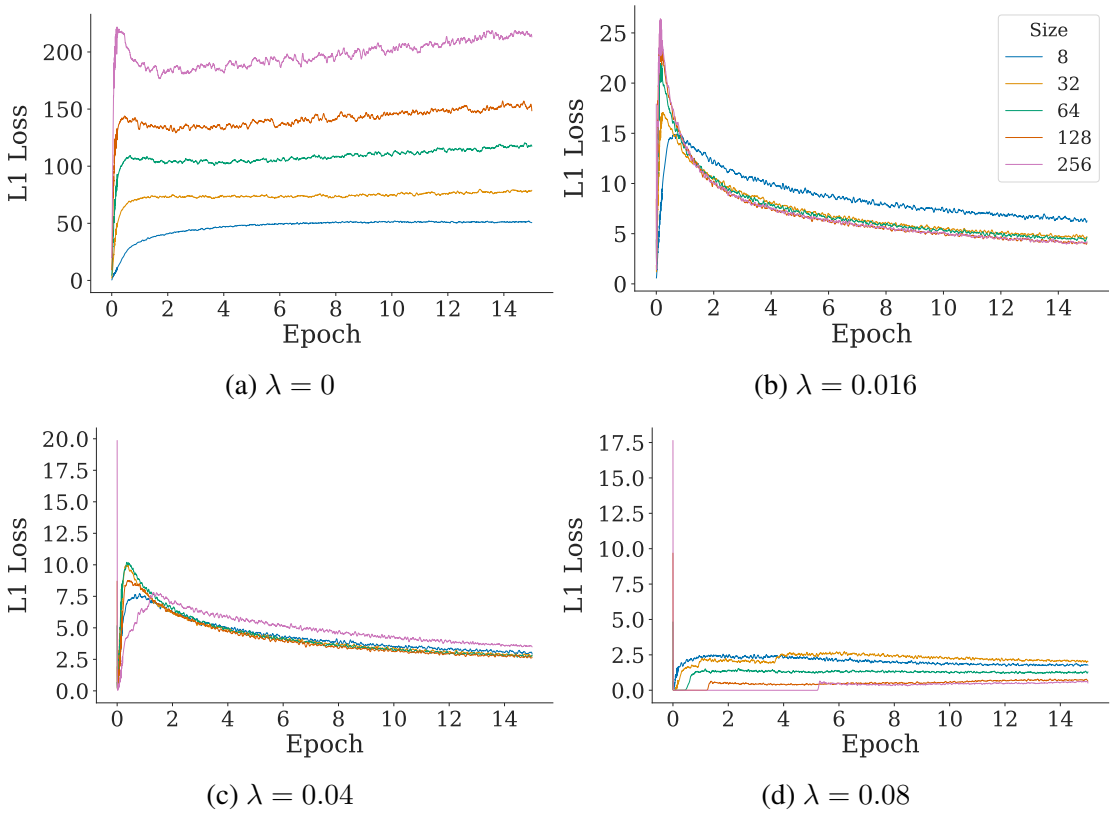


Figure 3.5: Average ℓ_1 activity per image (i.e., the sum of the absolute value of the activation values of all nodes, averaged over all test images) of single layer networks of various sizes. The decrease in ℓ_1 activity as λ increases shows that this method of activity regularisation does indeed decrease network activity.

The reason larger networks are more sensitive to activity regularisation is most likely because they just have more nodes, and hence are more likely to have a larger ℓ_1 activity. Figure 3.6a shows the ℓ_1 loss as training progresses in an unregularised network. Unsurprisingly, the largest networks had the largest levels of activity. At $\lambda = 0.016$ and greater, the 256 node network had a greater level of ℓ_1 activity early in the training process. As a result, there was a larger contribution to the gradient from the regularisation term for the larger networks early in training, so the larger networks became sparse more quickly. The larger gradient causes the sparsification to "overshoot" in larger networks compared to smaller networks, resulting in the largest networks ultimately having lower levels of ℓ_1 activity. Because the small networks begin with a small ℓ_1 activity, they are penalised to a much lesser extent early in training. As a result they become sparse on a longer time scale and do not "overshoot" in sparsity as the larger networks do.



(a) $\lambda = 0$ (b) $\lambda = 0.016$

(c) $\lambda = 0.04$ (d) $\lambda = 0.08$

Figure 3.6: An example of how the ℓ_1 loss for a network with 64 hidden nodes changes over the process of training.

3.1.3 Optimising the ℓ_1 Regularisation Coefficient, λ

There is some flexibility in what value one chooses for the regularisation coefficient λ, so long as it is not larger than the critical λ value described earlier. An approximate value can be easily characterised by simply training networks using a range of λ values and measuring their accuracy. The point at which accuracy begins to decrease can be taken as an upper bound for λ in later searches for an optimal value. Note also that it is not impossible for a network to achieve a reasonable accuracy at λ values greater than the critical value. In Figure 3.1 the maximum accuracy is far greater than the median for many values of $\lambda > \lambda_{crit}$, and occasionally one out of 20 networks trained with a large λ value would achieve an accuracy greater than the rest by $30 - 40$ percentage points. It is unclear however whether an accuracy comparable to an unregularised network can be achieved for $\lambda > \lambda_{crit}$, and even if it can be, the small gain in sparsity would not be worth the cost of training if only 1 in 20 networks or fewer was able to achieve such accuracy.

An optimal value of λ may also be determined by training networks with a range of λ values until one with the desired combination of sparsity and accuracy is found. Us-

ing this method qualitatively by inspecting Figures 3.1 and 3.2, it could be concluded that $\lambda = 0.008$ (or perhaps a value between 0 and 0.008) is a suitable regularisation coefficient. At this point accuracy is relatively unaffected, and larger values of λ do not increase sparsity by any significant amount.

A more quantitative method of determining an optimal λ might be to "weight" the accuracy of the network by a normalised measure of sparsity such as the Gini (Equation 1.18) or Hoyer index (Equation 1.19). This way, a network that is accurate but not sparse will be given a worse score than a network that performs well and uses few nodes for a given input. Figure 3.7 shows the average network accuracy weighted by the average Gini index, G, of the network.

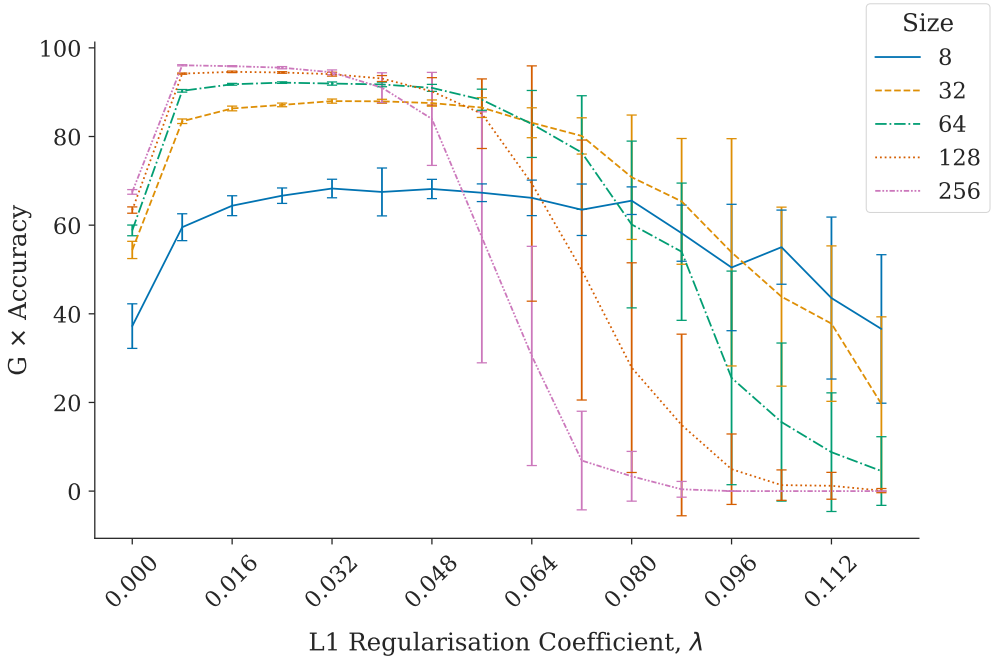


Figure 3.7: Average accuracy of each single layer ReLU network, weighted by its average Gini index, G. Both accuracy and Gini index are calculated as an average over 1000 images and 20 networks. The λ values corresponding to the maximum weighted accuracy for networks with 8, 32, 64, 128, and 256 hidden nodes respectively are as follows: 0.032, 0.032, 0.024, 0.016, 0.008.

While the accuracies for these values of λ may not offer the best performance, these values give a good balance between sparsity and accuracy. As will be seen later however, the optimal λ value depends not only on the network size, but on its specific architecture as well.

3.1.4 ReLU is Necessary for ℓ_0 Sparsity

Figure 3.8 shows the accuracy for various degrees of regularisation of single layer networks using the *sigmoid* activation funciton in hidden layers. While there is a drop in accuracy as λ is increased, the drop is not as dramatic as in ReLU networks, and testing suggests that this small decrease in accuracy can be almost entirely overcome with longer training times (these networks were only trained for 15 epochs).

These networks also had all nodes active at all times, regardless of the value of λ. This is unsurprising as the sigmoid function, $\sigma(x)$, asymptotically approaches zero as $x \to -\infty$, so it is not possible for a neuron with a sigmoid activation function to have an activation value of zero. Instead, regularisation has the effect of pushing activations towards small positive values. See Appendix A.1 to see example distributions of activation values in both ReLU and sigmoid fully connected networks.

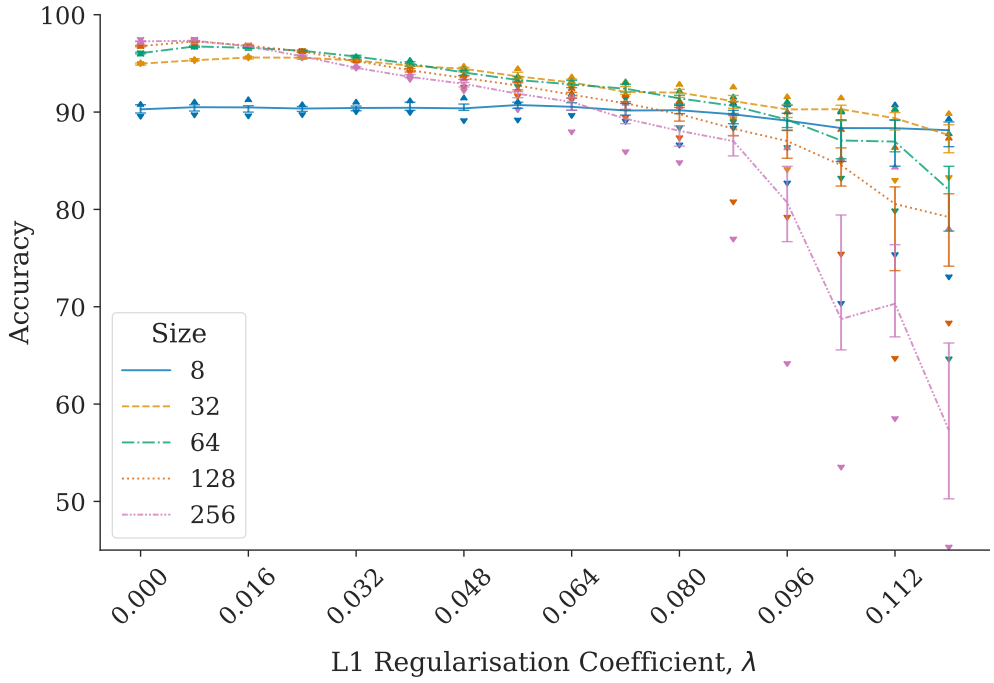


Figure 3.8: Median accuracy of five sizes of single layer fully connected networks using a sigmoid activation function, for different values of λ. Error bars show the inter-quartile range, upwards pointing triangles show the maximum accuracy, and downwards pointing triangles show the minimum accuracy.

Because sigmoid neurons cannot have an activation value of zero, such networks cannot be sparsely activated in the sense of having few active nodes. Its also clear that the ReLU activation function itself has sparsifying abilities. In a trained unregularised ReLU network, around $\%14$ of nodes in the 8 node network are inactive, and almost $\%50$ of nodes in the 256 node network are inactive. In comparison, in a sigmoid network nodes are

never inactive.

While sigmoid networks cannot be considered sparse when using the ℓ_0 norm as a measurement, they can be considered sparse when using other measures such as the Gini index. Figure 3.9 shows the Gini index for various sizes of networks with sigmoid activation functions. In this case a high level of sparsity means that the majority of the activity in the network is concentrated in a few nodes. In other words, a small proportion of nodes have large activation values, and a larger amount have small activation values.

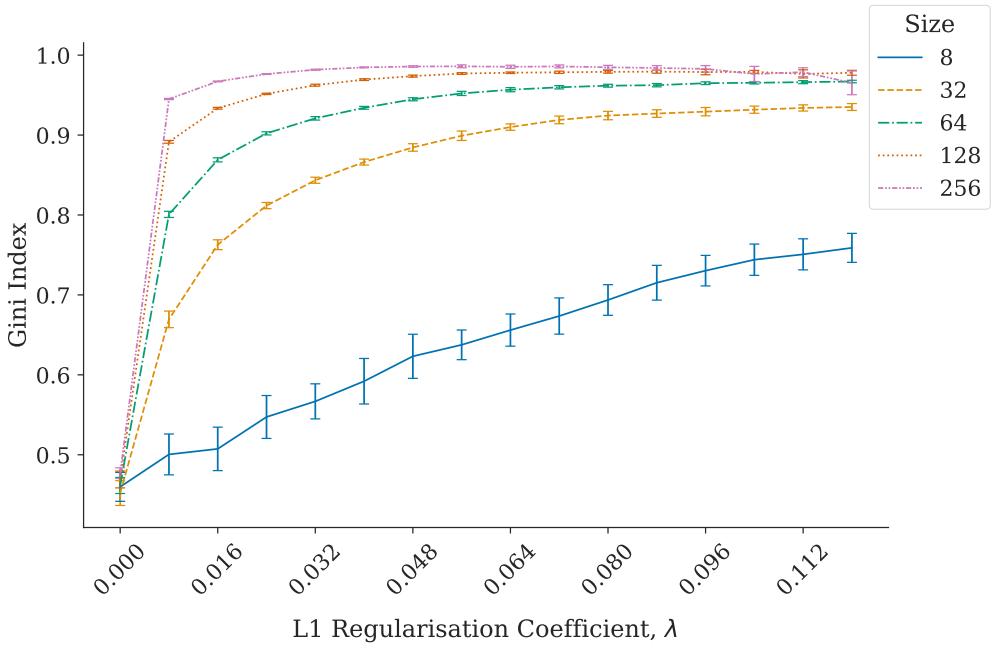


Figure 3.9: The average Gini index of single layer networks with sigmoid activation functions for various values of λ. While sigmoid networks cannot be sparse in the *number* of nodes that are active, they can be sparse in that a small proportion of nodes have a large activation value (close to 1), and a large proportion of nodes have relatively small levels of activity.

3.2 Deep Networks

3.2.1 Two Hidden Layers

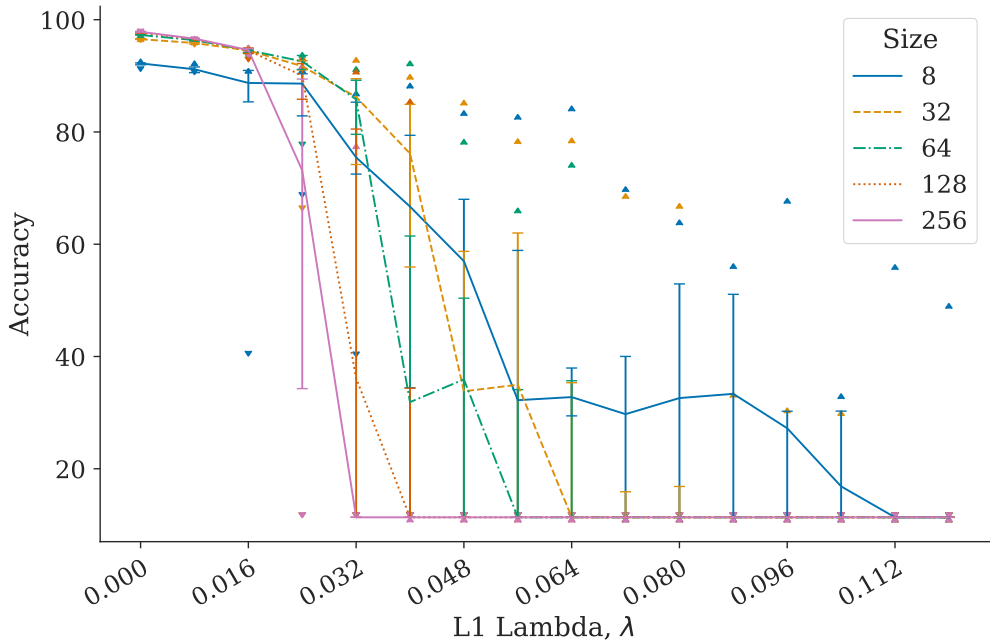
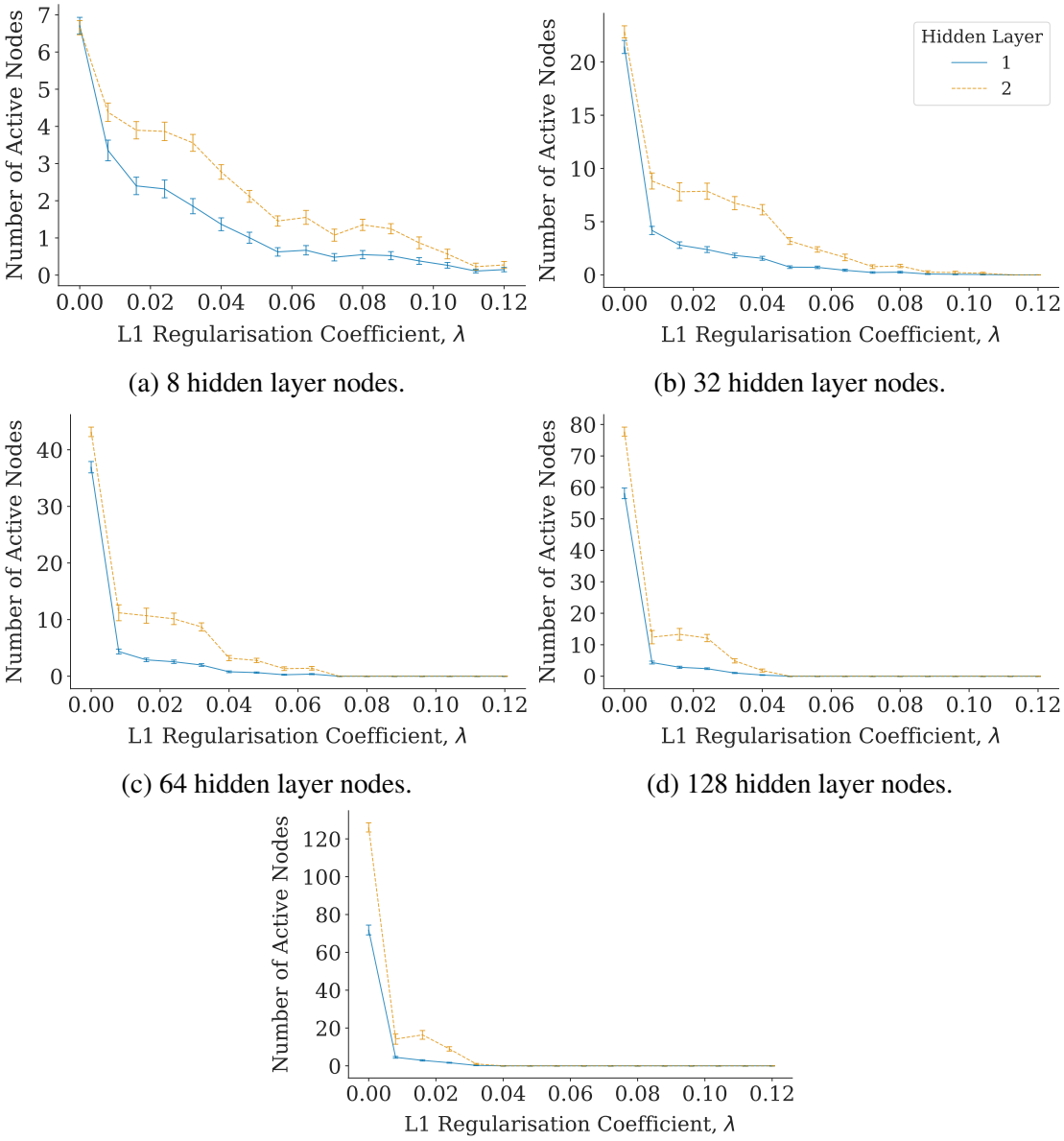


Figure 3.10: Median accuracy of five sizes of fully connected neural networks with two hidden layers, for different values of λ. Accuracy is defined as the proportion of correct predictions made by the network out of a set of test images. Error bars show the inter-quartile range, upwards pointing triangles show the maximum accuracy, and downwards pointing triangles show the minimum accuracy.

Figure 3.10 shows the accuracy of networks with two hidden layers for a variety of λ values. Following the same patterns as before, larger networks are more sensitive to regularisation than smaller networks. It follows that networks with more layers are also penalised more harshly than those with fewer layers.

Figure 3.11 shows the ℓ_0 activity in each layer of a network with two hidden layers. For all cases, the second hidden layer is more active than the first hidden layer. At first glance this is somewhat surprising - one might assume that if the inputs to a layer are sparse, then the layer itself will also be sparsely activated. Potential reasons for this will be discussed later.



(a) 8 hidden layer nodes.

(b) 32 hidden layer nodes.

(c) 64 hidden layer nodes.

(d) 128 hidden layer nodes.

(e) 256 hidden layer nodes.

Figure 3.11: Average number of active nodes in each layer of deep networks with two hidden layers. Five different sizes of hidden layer are shown here: 8, 32, 64, 128, and 256 nodes. Values were calculated by measuring the average number of nodes active per image for each network, and averaging over the 20 models trained for each test condition. Error bars indicate standard deviation.

3.2.2 Three Hidden Layers

Figure 3.12 shows the accuracy of a network with three hidden layers trained on MNIST for various values of λ. When unregularised, using three hidden layers does not offer a significant performance gain over single or two layer networks. As was the case for the two layer networks, three layer networks are even more sensitive to regularisation. As such, values of λ from 0 to 0.02 in steps of 0.001 were tested instead. While not as clear on this scale, a dramatic drop in accuracy still occurs at a critical λ value for the larger networks. As with the previous networks the decrease in accuracy is accompanied by an increase in the standard deviation of the accuracy, hinting that training is much less reliable when λ is too large.

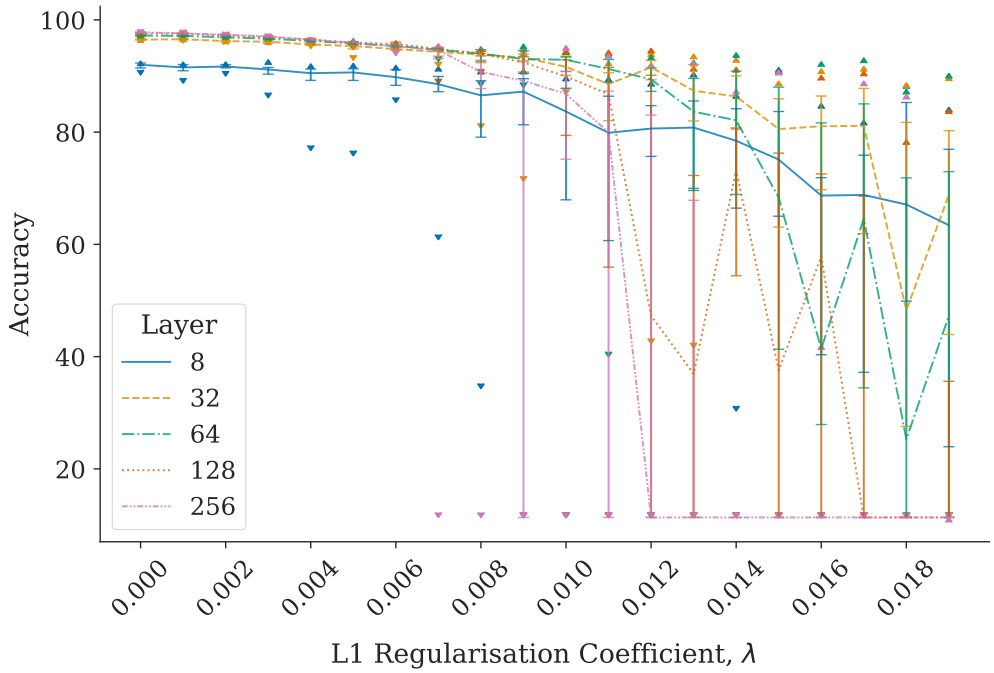
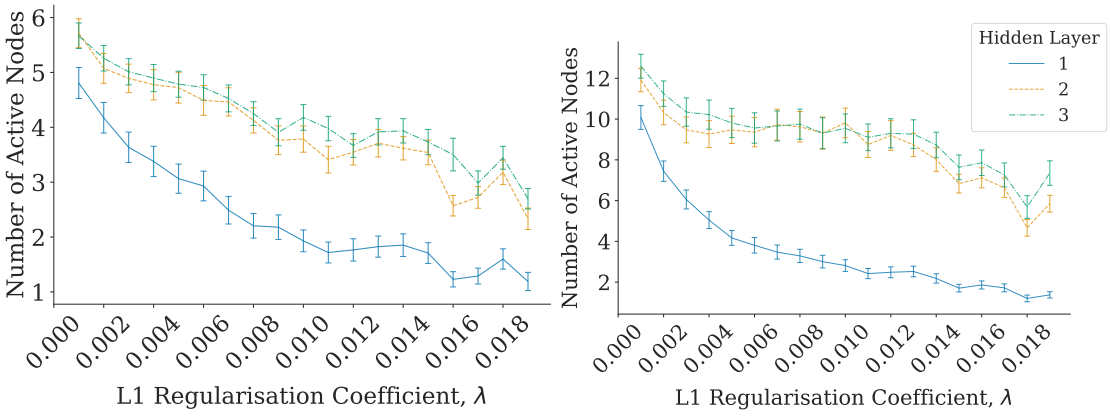
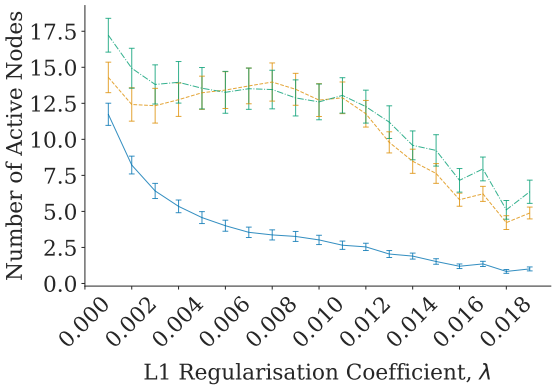


Figure 3.12: Median accuracy of five sizes of fully connected neural networks with three hidden layers, for different values of λ. Accuracy is defined as the proportion of correct predictions made by the network out of a set of test images. Error bars show the inter-quartile range, upwards pointing triangles show the maximum accuracy, and downwards pointing triangles show the minimum accuracy. Note that the range of λ values used here is smaller than those used in previous sections.

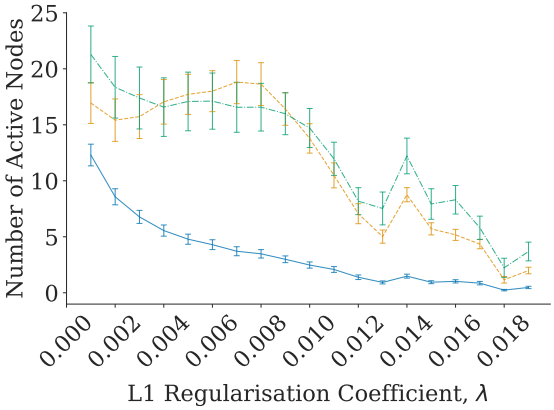


(a) 8 hidden layer nodes.

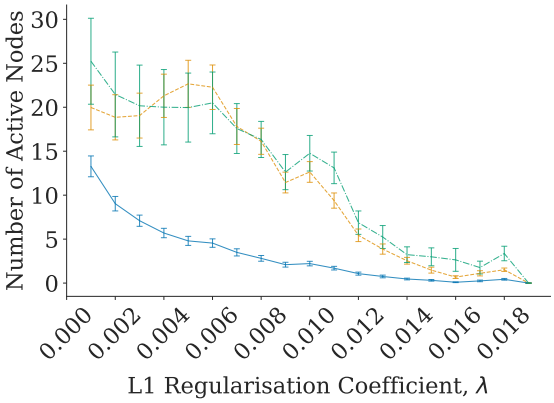
(b) 32 hidden layer nodes.



(c) 64 hidden layer nodes.



(d) 128 hidden layer nodes.



(e) 256 hidden layer nodes.

Figure 3.13: Average number of active nodes in each layer of deep networks with three hidden layers. Five different sizes of hidden layer are shown here: 8, 32, 64, 128, and 256 nodes. Values were calculated by measuring the average number of nodes active per image for each network, and averaging over the 20 models trained for each test condition. Error bars indicate standard deviation. Note that $\lambda = 0$ has been excluded from these graphs, and that the range of λ values shown here is smaller than those in previous sections.

Figure 3.13 shows the number of active nodes in each layer for the five sizes of three layer networks tested. While much smaller values of λ must be used to regularise these larger networks, it is still clearly possible to reduce the number of active nodes per layer without causing a large reduction in accuracy. As was seen in the two layer networks, the first hidden layer has fewer active nodes per image than subsequent layers. Both the second and third hidden layers have similar levels of activity. The first layer seems to be more sparse than the single layer networks for a given value of λ, however the later layers are less sparse when compared to the single layer network.

There are two potential reasons why the first layer in these deep networks may be more sparse than subsequent layers. First, a hidden layer will not necessarily be sparse just because its inputs are sparse because each node has an associated *bias* term. Recall that the activation value of a neuron is

$$a_i^l = f(\underset{\sim}{\boldsymbol{w}}_i^{(1)} \cdot \underset{\sim}{\boldsymbol{a}}^{(l-1)} + b_i^{(1)}),$$

where $b_i^{(1)}$ is the bias. This means that even if $\underset{\sim}{\boldsymbol{a}}^{(l-1)} = \underset{\sim}{\boldsymbol{0}}$, $\underset{\sim}{\boldsymbol{a}}^{(l)}$ may still be positive as long as $b_i^{(l)} > 0$. This is the case when a network is so regularised that all nodes are inactive for all inputs. The networks tested on the MNIST dataset reach a minimum accuracy of 11.35%. At this point all nodes in the network are dead, and the only inputs into the output layer are the output layer's biases. As a result, the network makes the same prediction every time regardless of the input. This occurs regardless of the number of hidden layers in the network, as the output layer always has associated bias terms.

As an alternative it may be useful to remove the bias terms from the network's output layer, and ensure that all other biases are strictly negative. By removing the final layer biases, they cannot be used to produce the same output for a dead network regardless of the input. Ensuring that the hidden layer biases are strictly negative would have a similar effect, but when used in tandem with the ReLU activation function the biases would act as a threshold. A neuron would then only produce an output when the input is greater than the bias, and if the neuron receives no inputs it will not produce an output. This is potentially more biologically realistic, as neurons have a threshold membrane potential that must be reached before an action potential (i.e., a signal) is produced. While this would stop a layer from being able to activate without non-zero inputs, it is unknown what the ultimate effect on sparsity and performance this change would have. This is outside the scope of this thesis, but of potential interest in future work.

To understand the second reason for a decrease in sparsity in later layers of the network, we need to understand the role that the regularisation term plays in backpropagation.

3.2.3 Backpropagation of the Regularisation Term

The second and possibly more significant reason as to why the first hidden layer in deep networks is more sparse than the later layers is due to the way in which the regularisation term contributes to the backpropagation equations. Recall that backpropagation refers to the process of finding the partial derivatives of the cost function with respect to the weights and biases of the network. We have seen the backpropagation equations for fully connected networks using the Cross Entropy cost function in Equation 1.34. Here we present the backpropagation equations for the activity regularisation term.

As an example, consider a network with two hidden layers where the activations of each layer are

$$a_i^{(2)} = f(\underset{\sim}{z}^{(2)}) = f(\underset{\sim}{w}_i^{(2)} \cdot \underset{\sim}{a}^{(1)} + b_i^{(2)}) \tag{3.2}$$

$$a_i^{(1)} = f(\underset{\sim}{z}^{(1)}) = f(\underset{\sim}{w}_i^{(1)} \cdot \underset{\sim}{x} + b_i^{(1)}) \tag{3.3}$$

$\underset{\sim}{w}_i^{(l)}$ is the row vector of weights connecting to the i^{th} neuron in layer l, and f is the ReLU activation function. Assume that the ReLU activation function is used for all further discussion in this chapter.

The ℓ_1 activity penalty is:

$$S = \sum_{l=1}^{2} \sum_{n} |a_n^{(l)}| \tag{3.4}$$

i.e., the sum over all activations in all hidden layers.

Consider then the partial derivatives of S with respect to $W_{ij}^{(2)}$ - the weight connecting *to* node i in layer 2 *from* node j in layer 1. In this network, only the second layer activations depend on these weights.

$$\begin{aligned}
\frac{\partial S}{\partial W_{ij}^{(2)}} &= \frac{\partial}{\partial W_{ij}^{(2)}} \left[\sum_n |a_n^{(2)}| \right] \\
&= \frac{\partial}{\partial W_{ij}^{(2)}} \left[f(\underset{\sim}{w}_i^{(2)} \cdot \underset{\sim}{a}^{(1)} + b_i^{(2)}) \right] \\
&= f'(\underset{\sim}{w}_i^{(2)} \cdot \underset{\sim}{a}^{(1)} + b_i^{(2)}) \cdot a_j^{(1)}
\end{aligned}$$

Where $f'(\underset{\sim}{z}^{(2)})$ is the derivative of the ReLU function

$$f'(x) = \begin{cases} 0, & \text{if } x < 0 \\ 1, & \text{if } x > 0 \\ \text{undefined}, & \text{if } x = 0 \end{cases} \tag{3.5}$$

$$\therefore \frac{\partial S}{\partial W_{ij}^{(2)}} = \begin{cases} 0, \text{ if } z_i^{(2)} < 0 \\ a_j^{(1)}, \text{ if } z_i^{(2)} > 0 \end{cases}$$

Consider now the weights leading into the first layer. Both the first and second layer activation values are functions of the first layer weights, so the partial derivative should include both terms. Hence the partial derivative with respect to the first layer weights is

$$\frac{\partial S}{\partial W_{ij}^{(1)}} = \frac{\partial}{\partial W_{ij}^{(1)}}\left[\sum_n |a_n^{(2)}| + \sum_n |a_n^{(1)}|\right]. \tag{3.6}$$

In layer one, only node $a_i^{(1)}$ depends on the weight $W_{ij}^{(2)}$. In layer two, all nodes are a function of $W_{ij}^{(2)}$. Hence Equation 3.6 becomes

$$\frac{\partial S}{\partial W_{ij}^{(1)}} = \frac{\partial}{\partial W_{ij}^{(1)}}\left[\sum_n |a_n^{(2)}| + a_i^{(1)}\right], \tag{3.7}$$

where

$$\begin{aligned}\frac{\partial}{\partial W_{ij}^{(1)}}\, a_i^{(1)} &= \frac{\partial}{\partial W_{ij}^{(1)}}\left[f(\underset{\sim}{\boldsymbol{w}}_i^{(1)}\underset{\sim}{\boldsymbol{x}} + b_i^{(1)})\right] \\ &= f'(\underset{\sim}{\boldsymbol{w}}_i^{(1)}\underset{\sim}{\boldsymbol{x}} + b_i^{(1)}) \cdot x_j\end{aligned}$$

and

$$\begin{aligned}\frac{\partial}{\partial W_{ij}^{(1)}} \sum_n |a_n^{(2)}| &= \frac{\partial}{\partial W_{ij}^{(1)}}\left[\sum_n f(\underset{\sim}{\boldsymbol{w}}_n^{(2)}\underset{\sim}{\boldsymbol{a}}^{(1)} + b_n^{(2)})\right] \\ &= \sum_n f'(\underset{\sim}{\boldsymbol{w}}_n^{(2)}\underset{\sim}{\boldsymbol{a}}^{(1)} + b_n^{(2)}) \cdot W_{ni}^{(2)} \cdot \frac{\partial}{\partial W_{ij}^{(1)}}\, a_i^{(1)} \\ &= \sum_n f'(z_n^{(2)}) W_{ni}^{(2)} f'(z_i^{(1)}) x_j\end{aligned}$$

In summary, we have

$$\frac{\partial S}{\partial W_{ij}^{(2)}} = \begin{cases} 0, \text{ if } z^{(2)} < 0 \\ a_j^{(1)}, \text{ if } z^{(2)} > 0 \end{cases} \tag{3.8}$$

and when $z_i^{(1)}$ is positive,

$$\frac{\partial S}{\partial W_{ij}^{(1)}} = \sum_n f'(z_n^{(2)}) W_{ni}^{(2)} x_j + x_j \tag{3.9}$$

Now recall from Equation 1.33, during gradient descent a weight is changed by an amount proportional to its gradient. For the second layer weights, the portion of the gradient contributed by the regularisation term depends only on the activity of the first layer. For the first layer weights however, both the input to the network and the second layer weights contribute.

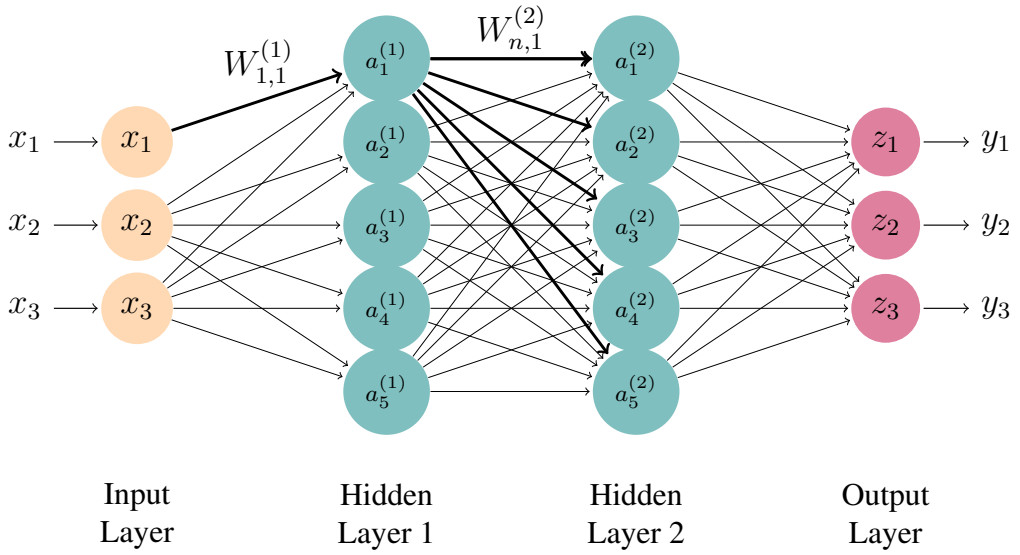


Figure 3.14: A feedforward neural network with two hidden layers showing the back-propagation path taken to calculate the partial derivative of R with respect to the first layer weight $W^{(1)}_{1,1}$. Every node in layer 2 depends on this weight, but only the first node in layer 1 is a function of the weight. As a result, the weights connecting all nodes in layer 2 to node 1 in layer 1 contribute to the partial derivative with respect to $W^{(1)}_{1,1}$. (Created using code adapted from TikZBlog [33])

In *PyTorch*, the weights for each fully connected layer are initialised by drawing them from a uniform distribution $\mathscr{U}(-\sqrt{k}, \sqrt{k})$ with a mean of 0, where $k = \frac{1}{\text{in features}}$ (in features is the number of inputs to the layer). We can assume then that in an untrained network the first term in Equation 3.9 will, on average, be close to zero, as there will be approximately an equal amount of positive and negative weights. As an example, from a sample of 10000 randomly initialised networks the average sum of weights from node 1 in layer 1 to all nodes in layer 2, $\sum_n W^{(2)}_{n,1}$, was 0.008 ± 0.573. As a result, Equation 3.9 will be dominated by the second term, x_j, which denotes a pixel in the input image and can take on values between 0 and 1.

In comparison, the portion of the gradient contributed by the second layer weights depend only on the activations in the first layer, which will in general be small in an untrained network. As a result, at the beginning of training when the network is essentially untrained the regularisation penalty will be larger for the weights in the first layer than in the second layer, causing the first layer weights to evolve towards a set of weights that result in sparseness much quicker. Furthermore, once the first layer becomes sparsely activated and many first layer nodes become inactive, many second layer weights will no longer be affected by activity regularisation as the portion of their gradient contributed by regularisation only depends on the activation value of the first layer node they connect from. As a result, this method of regularisation is intimately connected to the way in which the network learns. This decreases its validity as a method of exploring the

role of sparsity across layers of the visual system. The way in which a model of sparsity in the brain generates sparsity should reflect the ways sparsity is created in the brain, or else at least be capable of encompassing potential mechanisms. The linear coding models presented in Section 1.3 achieve this via a notion of "lateral connections" between nodes which introduces competition between nodes, as well as self inhibition. This allows the optimal coefficients to be selected via a "local learning" process.

In the case of fully connected networks, however, sparsity is created by shifting the distribution of weights in the network towards negative values. Negative inputs to ReLU produce an output of 0, and negative inputs to sigmoid produce small positive outputs. If the distribution of weights in the network is negatively skewed, the input to each node is itself more likely to be negative, resulting in a greater number of small or zero activation values once applying the activation value. This could be interpreted as *inter-layer* inhibition, where nodes in one layer inhibit the nodes in the following layer. Figures A.1 and A.2 in Appendix A.1 show the distribution of activation values from 1000 images in a 64 node single layer fully connected network, for a range of λ values.

Further, classification algorithms such as the fully connected networks shown here are *discriminative* models. Discriminative models model the conditional probability of an outcome given the data, $P(Y|X)$. In comparison, the linear coding models in Section 1.3 are *generative* models. Generative models model the joint distribution of the data and the outcomes, $P(X, Y)$. In the case of visual perception, the data X are the visual signals received by the retina, and the "outcomes" are instead the brain's explanations of its environment, which are represented in the neural code, or the coefficients a_i. If visual perception is in fact a process of statistical inference, the probabilistic model that the brain would form to perform this inference would almost certainly be generative, not discriminative. Discriminative models are used for classification – they map an input to a category. The brain, however, must do much more than simple classification on the signals it receives, and as discussed in Chapter 1, perception is an ill-posed problem and there is not a simple one-to-one mapping from the data to an understanding of the environment.

Evidence suggests that sparsity likely increases across layers of the visual system as the information represented by neurons becomes more specific [44, 45]. While this result makes any concrete conclusions about sparsity in the visual system difficult, it appears to be a little studied topic in deep learning, and helps to highlight the care that must be taken when creating models of visual processing.

CHAPTER 4

Convolutional Networks and Autoencoders

Given the results of the previous section, we now wish to know how sparsity behaves in other network architectures. First, this will allow us to determine whether the process of sparsification depends on the networks architecture, or whether the method of inducing sparse activity used here works the same regardless of architecture. Second, it may give different insights into how a sparse network behaves, as convolutional networks and autoencoders learn features differently to fully connected networks

4.1 Convolutional Networks

Convolutional networks with 1, 2, and 3 convolutional layers were trained and tested. In each case, kernel sizes of 3×3, 7×7, and 11×11 were used, with the same kernel size used in each layer (so a total of 9 architecture variations were tested). For all networks 6 kernels were used in the first convolutional layer, 16 in the second, and 24 in the third, with a final fully connected layer with 64 nodes connecting to a 10 node output layer. A summary of the architectures used is shown in Table 4.1, and an example diagram of a CNN with two convolutional layers is shown in Figure 1.6.

Convolutional networks were trained and tested on both the MNIST and CIFAR-10 datasets.

Table 4.1: A summary of the convolutional network architectures presented in this section. Note that 3 layer networks with 3×3 kernels were note used for the MNIST dataset.

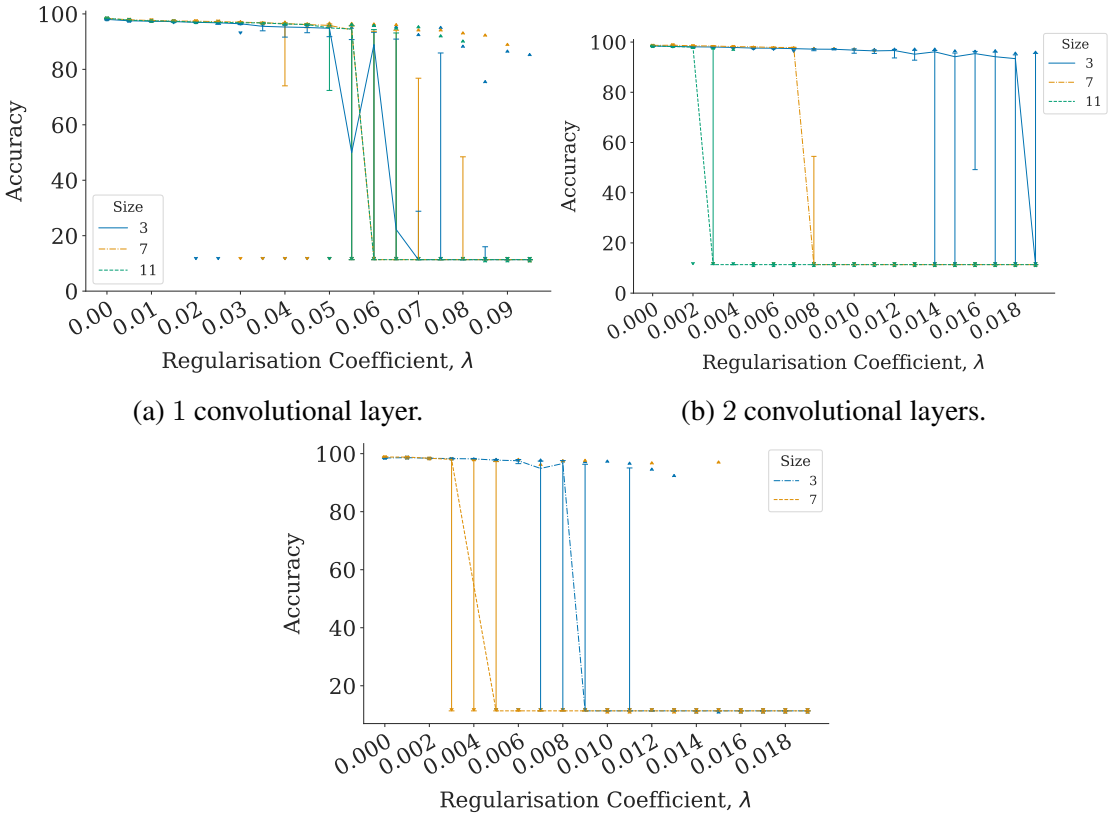
Number of layers	kernel sizes	Layer 1 channels	Layer 2 channels	Layer 3 channels
1	3×3, 7×7, 11×11	6	–	–
2	3×3, 7×7, 11×11	6	16	–
3	3×3, 7×7, 11×11	6	16	24

4.1.1 MNIST

As with previous architectures, convolutional networks were trained and tested on the MNIST dataset of handwritten digits. As should be expected, convolutional networks of all sizes consistently out-performed fully connected networks at classification of handwritten digits. Again, a critical λ value at which accuracy catastrophically decreases was observed in all CNNs. The same trade off between network accuracy and sparsity could be seen in all CNNs tested, and it was again clearly possible to reduce the number of active nodes in the network without a significant hit to performance.

In Chapter 3, networks with more nodes were more sensitive to regularisation. However, in this case, CNNs with larger kernel sizes seemed to be more sensitive to regularisation than those using smaller kernel sizes. Using a larger kernel results in a CNN with fewer nodes, since larger kernels result in greater dimensionality reduction in the data, and hence fewer nodes in the following layers. For example, using Equation 1.41, if a 3×3 kernel is used with a stride length of 1 and no padding, then one set of convolutions on an MNIST image would result in a 26×26 image (i.e 676 nodes in the first convolutional layer), and a second round would result in a 24×24 image (575 nodes in the second convolutional layer). By comparison, if a 11×11 kernel is used, this reduces to a 16×16 image (i.e 256 nodes) in the first layer, and a 6×6 image (36 nodes) in the second layer - far fewer nodes than in the 3×3 case. Despite having fewer nodes, CNNs using larger kernels appeared to be more sensitive to regularisation.

Figures 4.2 and 4.3 show the average number of active nodes in the convolutional layers of CNNs with two and three convolutional layers respectively. In each case, the number of active nodes related closely to the network accuracy in Figure 4.1. The point at which accuracy dropped off was roughly in line with the point at which the number of active nodes dropped to near zero. In the case of the 3×3 kernel, the number of active nodes did not drop to zero, but instead levelled out at around 250 active nodes.



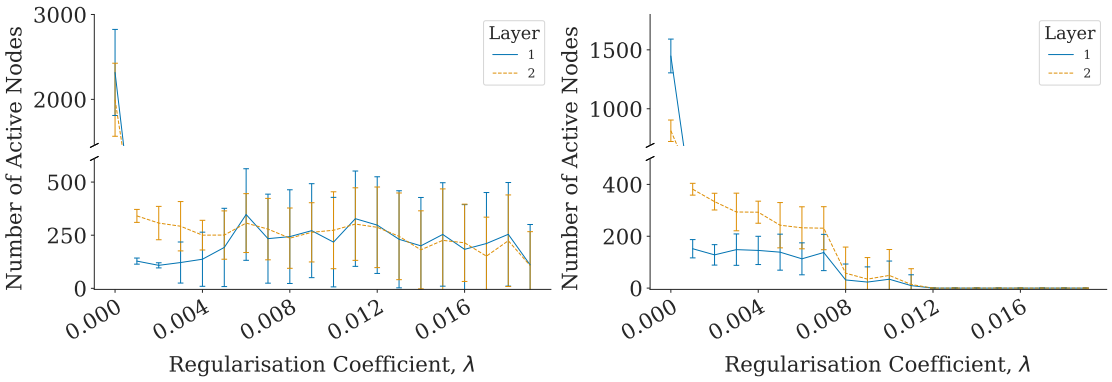
(a) 1 convolutional layer.

(b) 2 convolutional layers.

(c) 3 convolutional layers.

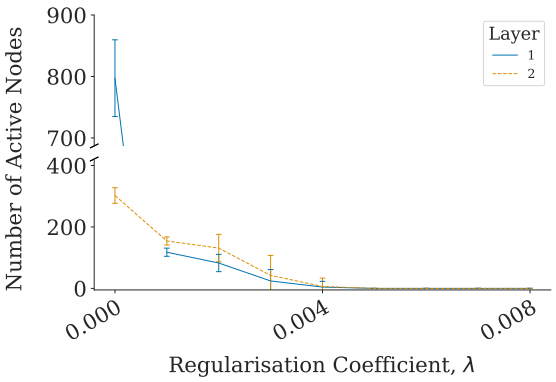
Figure 4.1: Median accuracy of convolutional neural networks trained on MNIST, for different values of λ. Accuracy is defined as the proportion of correct predictions made by the network out of a set of test images. Error bars show the inter-quartile range, upwards pointing triangles show the maximum accuracy, and downwards pointing triangles show the minimum accuracy. Note also the different λ scale in Subfigure 4.1a

The difference in sparsity across layers that was visible in fully connected networks was also observed in CNNS, although it was not as clear. In Figures 4.2b, 4.2c, and 4.3a sparsity clearly decreased in each successive layer (i.e., the number of active nodes increases). In Figures 4.3b and 4.2a however the difference in activity across layers was much smaller. Note, however, that the total number of nodes in each layer depends not only on the size of the kernel being used, but also on the number of kernels used in each layer. As a result of the architectures used here, each successive layer ends up containing more nodes than the previous layer. Regardless, regularisation still had the effect of considerably decreasing the number of active nodes, with minimal effect on accuracy.



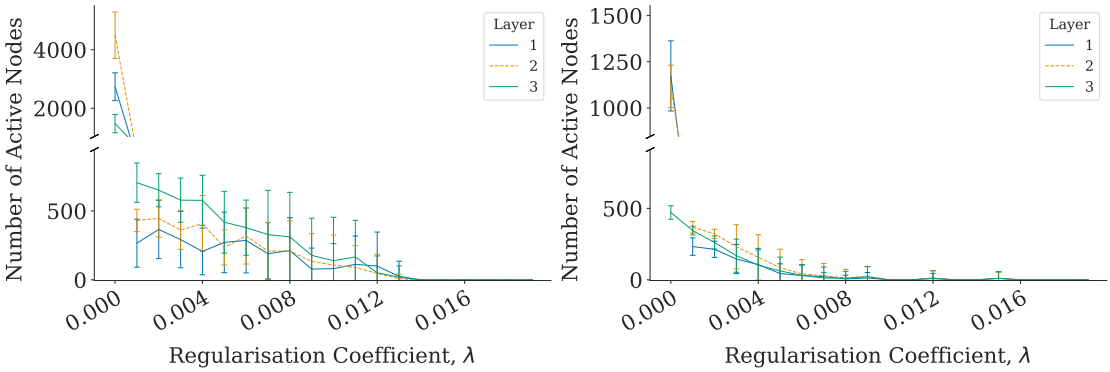
(a) 3×3 kernel.

(b) 7×7 kernel.



(c) 11×11 kernel.

Figure 4.2: Average number of active nodes per image in each convolutional layer of two layer CNNs trained on MNIST, with (a) 3×3, (b) 7×7 , and (c) 11×11 sized kernels. Values were calculated by measuring the average number of nodes active per image for each network, and averaging over the 20 models trained for each test condition. Note the broken y axis on each plot. Note also that plot (c) covers a smaller range of λ values.



(a) 3×3 kernel.

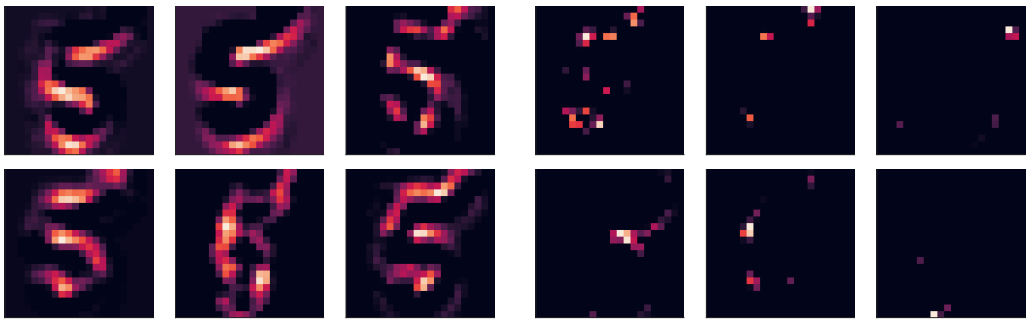
(b) 7×7 kernel.

Figure 4.3: Average number of active nodes per image in each convolutional layer of three layer CNNs trained on MNIST, with (a) 3×3 , and (b) 7×7 sized kernels. Values were calculated by measuring the average number of nodes active per image for each network, and averaging over the 20 models trained for each test condition. Note the broken y axis on each plot. 11×11 kernels could not be used in the three layer CNNs, as after two convolutions the feature maps would be 8×8 in size - smaller than the kernel.

Figure 4.4 shows an example of the feature maps (an image showing the activation value of each node) from a two layer CNN when the image in Figure 4.5 is used as the input. On the left are the first and second layer feature maps for an unregularised network, and on the right are the feature maps for a CNN regularised with $\lambda = 0.002$. At this λ value, accuracy was still largely unaffected (see Figure 4.1b), and there were clearly far fewer active nodes than in the unregularised network.

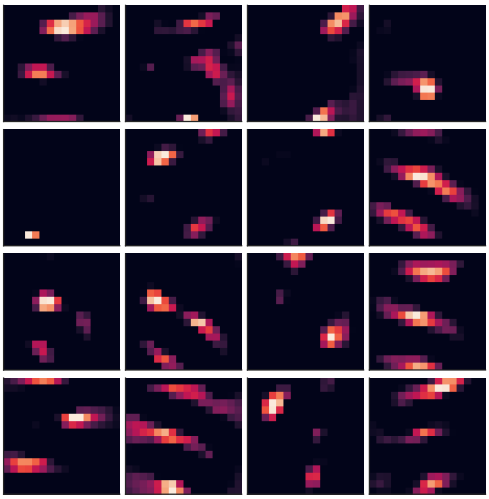
The first layer feature maps in the unregularised network were all qualitatively similar to one another. Some showed the outline of the digit, while others still strongly resembled the digit itself. In the second layer the features were further processed into clear sets of lines. In comparison, the first layer feature maps in the regularised network no longer resembled the input digit as clearly, instead containing far fewer active neurons. Additionally, the feature maps were no longer similar to one another. Some were largely inactive, while the remaining feature maps represented different parts of the input digit. This continued in the second layer feature maps, in which at least six appeared entirely inactive, and the clear structure seen in the unregularised feature maps was not present.

This suggests that there is redundant information encoded in the feature maps of the unregularised networks which is not necessarily needed to classify a given digit. It is clearly not necessary to extract multiple similar features from a digit in order to classify it. Rather, only a few, sparse features are needed. If the regularised networks are extracting different features, they must be learning different kernels.

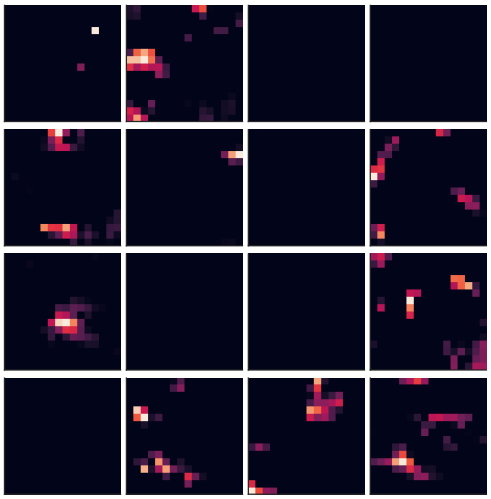


(a) Layer one feature maps for $\lambda = 0$.

(b) Layer one feature maps for $\lambda = 0.002$.



(c) Layer two feature maps for $\lambda = 0$.



(d) Layer two feature maps for $\lambda = 0.002$.

Figure 4.4: Example node activity in layers one (top) and two (bottom) of an unregularised convolutional network (left) and a regularised network (right, $\lambda = 0.002$) with two convolutional layers and 7×7 convolution kernels trained on MNIST. The digit in Figure 4.5 was used as the input for these examples.

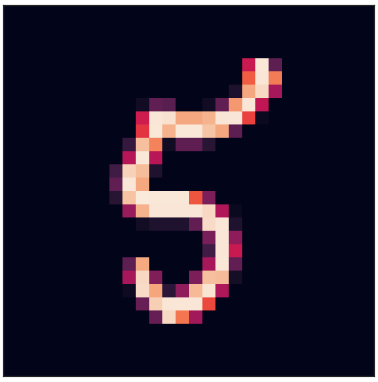
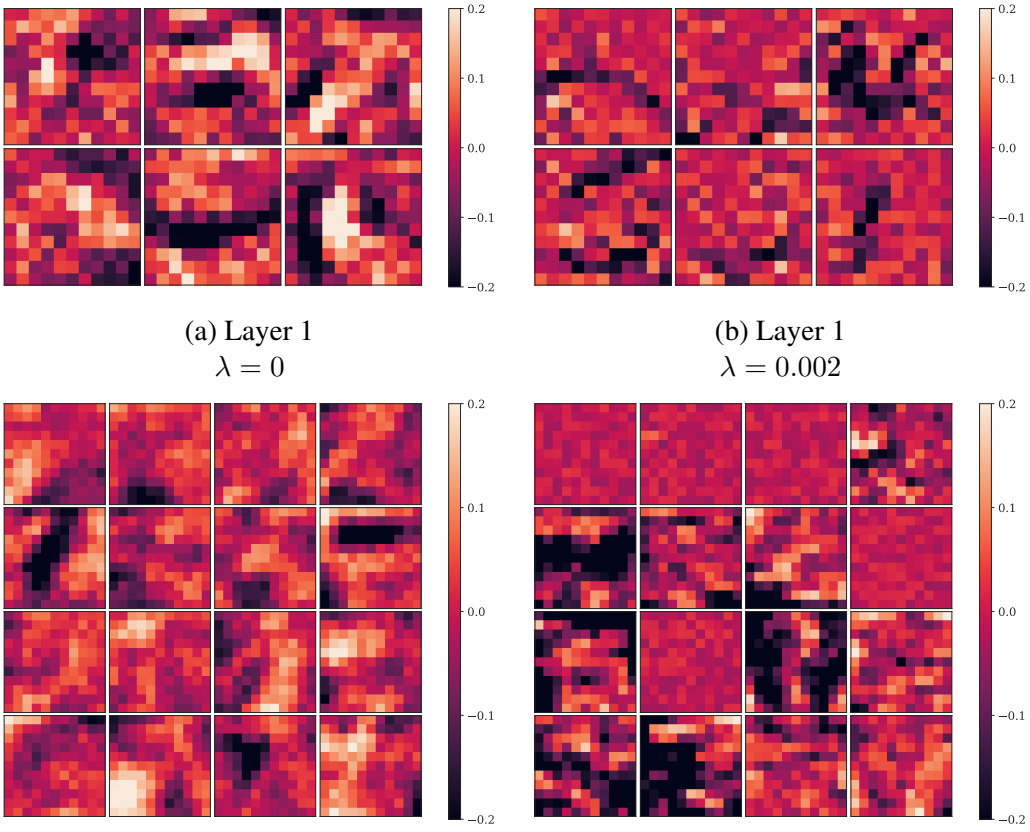


Figure 4.5: An example digit "5" from the MNIST dataset, used as the input for the example feature maps in Figure 4.4

Figure 4.6 shows example 11×11 kernels from a CNN with two convolutional layers, with figures on the left showing those for $\lambda = 0$, and figures on the right showing those for $\lambda = 0.002$. Some structure could be seen in the unregularised kernels, with some containing alternating light and dark regions. This structure was less clear in the regularised kernels, and any similar structures appeared much more localised (e.g. dark lines were a single pixel wide rather than two or three). There also appear to be some second layer kernels with no structure at all, indicating that these kernels have undergone very little learning and do not play a significant role in the feature extraction process.



(a) Layer 1
$\lambda = 0$

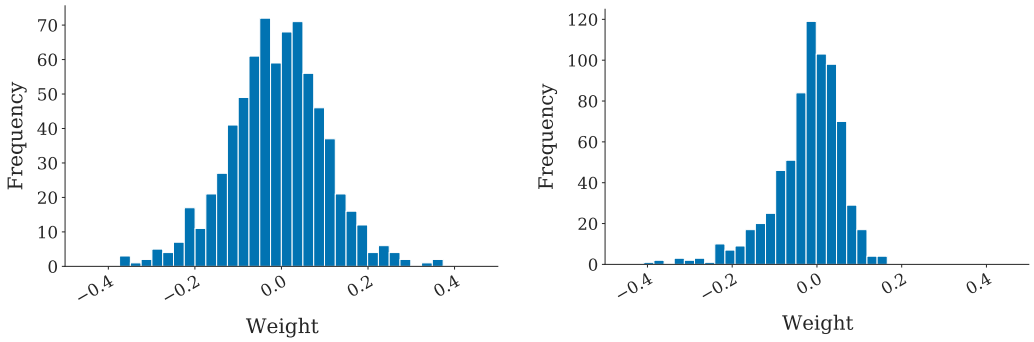
(b) Layer 1
$\lambda = 0.002$

(c) Layer 2
$\lambda = 0$

(d) Layer 2
$\lambda = 0.002$

Figure 4.6: Examples of 11×11 kernels from CNNs using two convolutional layers trained on MNIST for $\lambda = 0$ and $\lambda = 0.002$. The top two plots show the 6 kernels used in the first convolutional layer for each value of λ, and the bottom two plots show the 16 kernels from the second layer. The particular networks whose kernels are shown in this figure were chosen as an example as some structure is still visible in most kernels, while in some other networks trained with the same parameters all kernels appeared to be lacking in structure.

Kernels in regularised networks also appeared to contain more negative weights than those of unregularised networks. This can be seen more clearly in Figure 4.7, which shows the distribution of weights from the first layer kernels of the same network shown in Figure 4.6. In the regularised networks there was a clear negative skew in the distribution of weights. This makes sense, given that ReLU was used as the activation function. Recall that $ReLU(x) = \max(0, x)$, hence the more negative the weights are connecting to a neuron, the less likely it is that the neuron will activate. A similar effect on the weights was seen in fully connected networks.



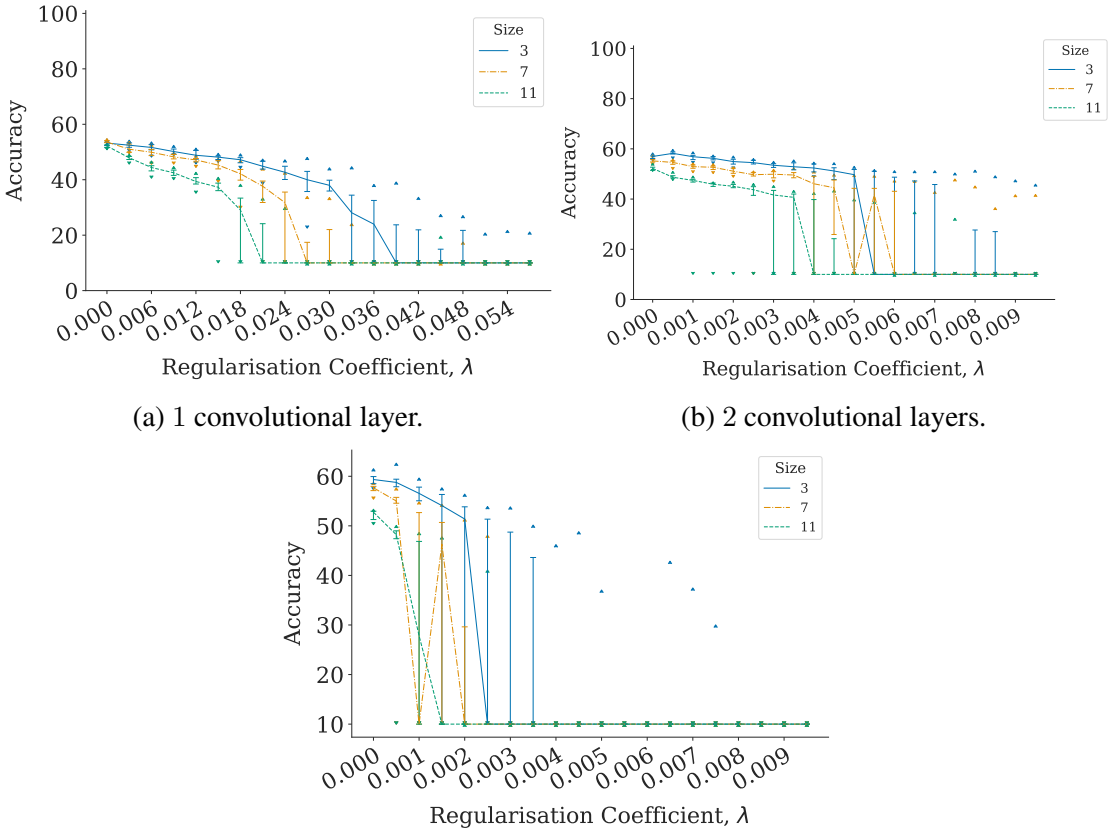
(a) Layer 1 weights for $\lambda = 0$

(b) Layer 1 weights for $\lambda = 0.002$

Figure 4.7: Distribution of kernel weights from the first layer Kernels of a convolutional network with two convolutional layers. Subfigure (a) shows the weights for an unregularised network, and Subfigure (b) shows the weights for a network regularised with $\lambda = 0.002$.

4.1.2 CIFAR-10

The same set of CNN architectures were also trained and tested on the CIFAR-10 dataset, with similar results. CIFAR-10 is a much more challenging dataset to perform classification on, as the images are far more complex than MNIST digits. As such, these networks performed much more poorly. Figure 4.8 shows the median accuracy for these networks. Figure 4.9 and 4.10 show the average number of active nodes per image in each layer of two and three layer CNNs respectively. Again in most cases the first layer was the most sparse, with sparsity decreasing in successive layers. This decrease was also less clear in three layer CNNs. In Figure 4.10c, the final layer was by far the least active. This was a result of three layers of convolutions using an 11×11 kernel, resulting in the third layer consisting of 24 channels of 2×2 feature maps (a total of 96 nodes – far fewer than preceding layers).

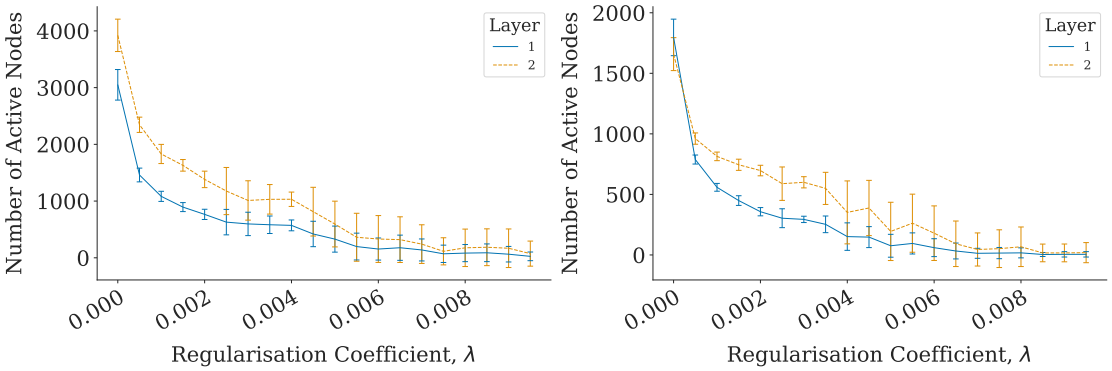


(a) 1 convolutional layer.

(b) 2 convolutional layers.

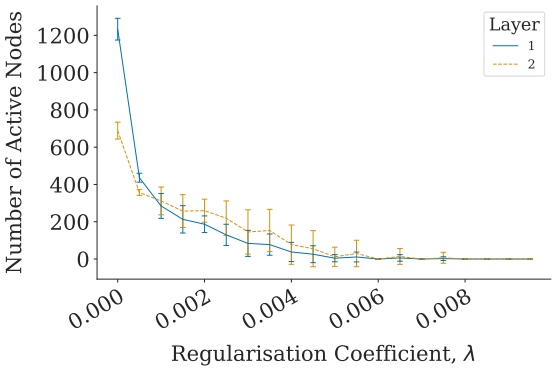
(c) 3 convolutional layers.

Figure 4.8: Median accuracy of convolutional neural networks trained on CIFAR-10, for different values of λ. Accuracy is defined as the proportion of correct predictions made by the network out of a set of test images. Error bars show the inter-quartile range, upwards pointing triangles show the maximum accuracy, and downwards pointing triangles show the minimum accuracy.



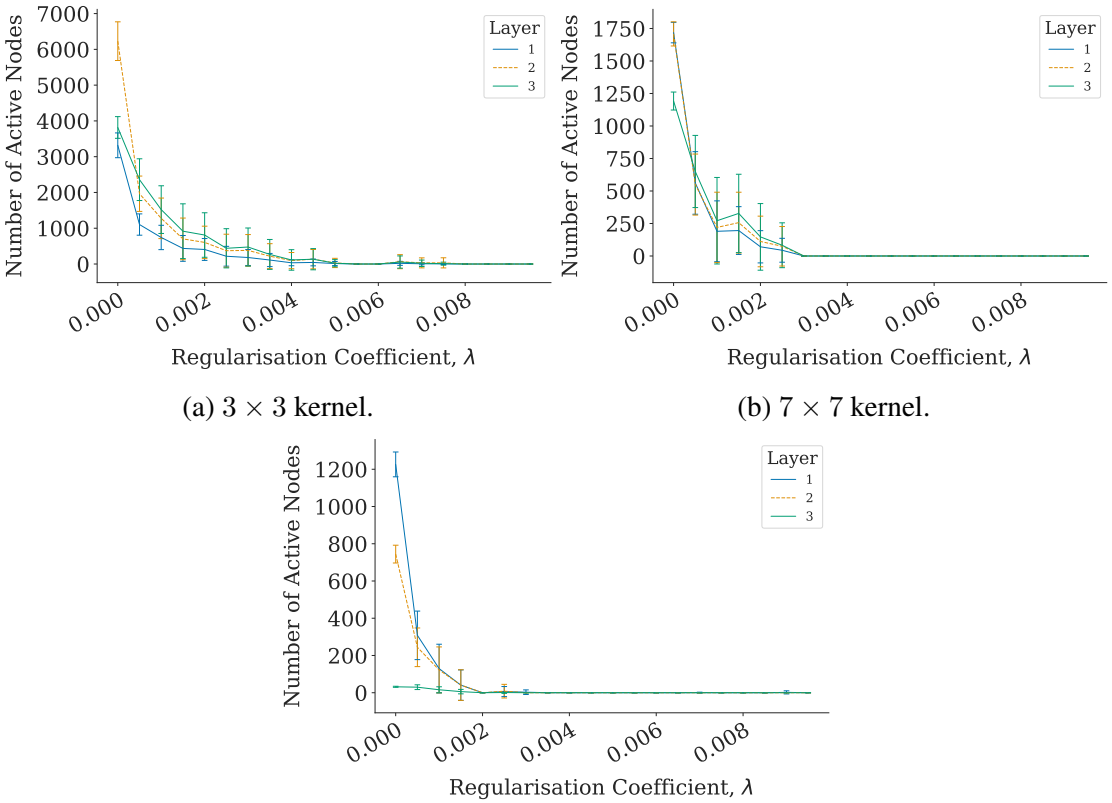
(a) 3×3 kernel.

(b) 7×7 kernel.



(c) 11×11 kernel.

Figure 4.9: Average number of active nodes per image in each convolutional layer of two layer CNNs trained on CIFAR-10, with (a) 3×3, (b) 7×7 , and (c) 11×11 sized kernels. Values were calculated by measuring the average number of nodes active per image for each network, and averaging over the 20 models trained for each test condition. Error bars indicate standard deviation.



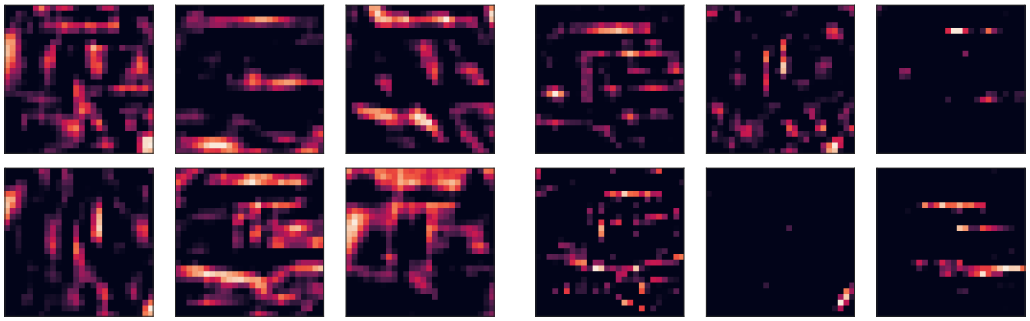
(a) 3×3 kernel.

(b) 7×7 kernel.

(c) 11×11 kernel.

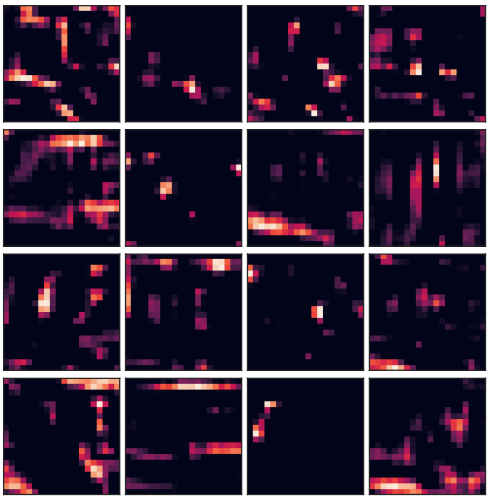
Figure 4.10: Average number of active nodes per image in each convolutional layer of three layer CNNs trained on CIFAR-10, with (a) 3×3, (b) 7×7 , and (c) 11×11 sized kernels. Values were calculated by measuring the average number of nodes active per image for each network, and averaging over the 20 models trained for each test condition. Error bars indicate standard deviation.

Figure 4.11 shows the feature maps (active nodes) from the first (top) and second (bottom) convolutional layers of an unregularised (left) and a regularised (right) CNN. As with the MNIST dataset, there is a clear reduction in the number of active nodes in both layers, and many feature maps are almost entirely empty. Again, as in the MNIST dataset, regularisation appeared to have the effect of reducing the redundancy in the features that were extracted.

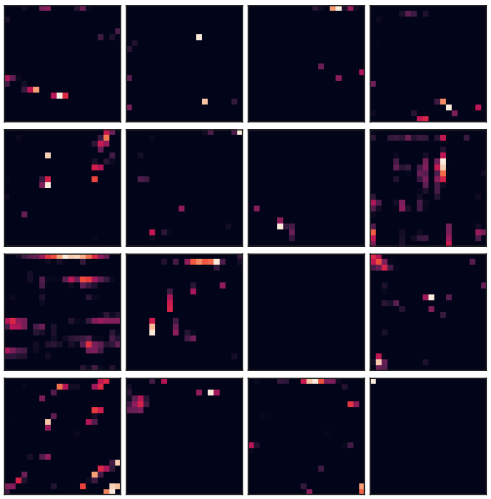


(a) Layer one feature maps for $\lambda = 0$.

(b) Layer one feature maps for $\lambda = 0.002$.



(c) Layer two feature maps for $\lambda = 0$.

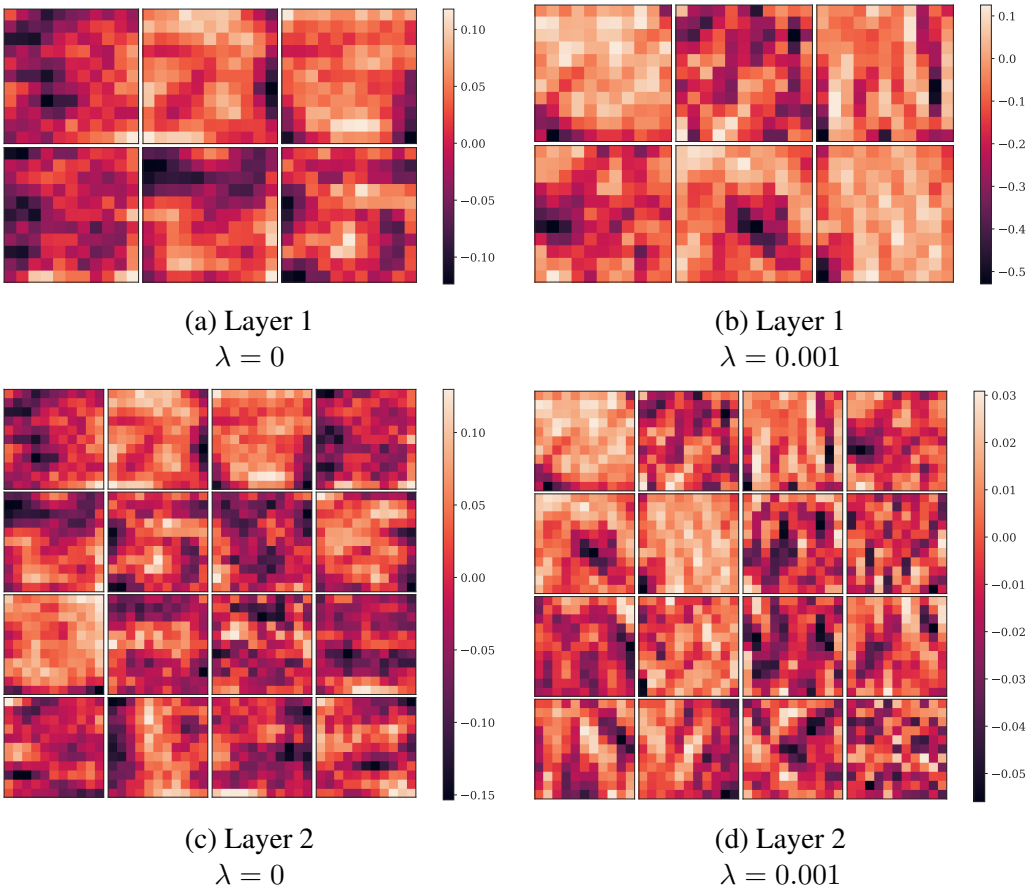


(d) Layer two feature maps for $\lambda = 0.002$.

Figure 4.11: Example node activations in layers one (top) and two (bottom) of an unregularised convolutional network (left) and a regularised network (right, $\lambda = 0.002$) with two convolutional layers and 7×7 convolution kernels trained on CIFAR-10. The input for this example is shown in Figure 4.12.

Figure 4.12: An example truck from the CIFAR-10 database.

Figure 4.13 shows the first and second layer kernels from a two layer CNN with 11×11 kernels trained on CIFAR-10. Qualitatively there appeared to be a difference between the kernels of regularised and unregularised networks. A more thorough investigation into the structure of the learnt kernels would be needed to determine the effects of sparsity on these kernels.



(a) Layer 1
$\lambda = 0$

(b) Layer 1
$\lambda = 0.001$

(c) Layer 2
$\lambda = 0$

(d) Layer 2
$\lambda = 0.001$

Figure 4.13: Examples of 11×11 kernels from CNNs trained on CIFAR-10 using two convolutional layers for $\lambda = 0$ and $\lambda = 0.002$. The top two plots show the 6 kernels used in the first convolutional layer for each value of λ, and the bottom two plots show the 16 kernels from the second layer. The particular networks whose kernels are shown in this figure were chosen as an example as some structure is still visible in most kernels, while in some other networks trained with the same parameters all kernels appeared to be lacking in structure.

4.2 Autoencoders

Autoencoders with three hidden layers were trained on the MNIST dataset – one hidden encoding layer, one hidden decoding layer, and the central "code" layer. Three different sizes were trained:

- $784 \to 400 \to 64 \to 400 \to 784$
- $784 \to 784 \to 784 \to 784 \to 784$
- $784 \to 900 \to 1024 \to 900 \to 784$

corresponding to input size $\to$ encoding layer $\to$ code layer $\to$ decoding layer $\to$ output size. By doing so the traditional bottleneck architecture could be compared to architectures in which the number of nodes in the "code" layer was greater than or equal to the dimension of the input. All hidden layers in the autoencoders were regularised as in Equation 2.4, with λ ranging from 0 to 9×10^{-4} in steps of 0.5×10^{-4}. The goal of this is to determine whether sparsity affects autoencoder performance, and to see whether an overcomplete autoencoder can learn a sparse representation of the input data.

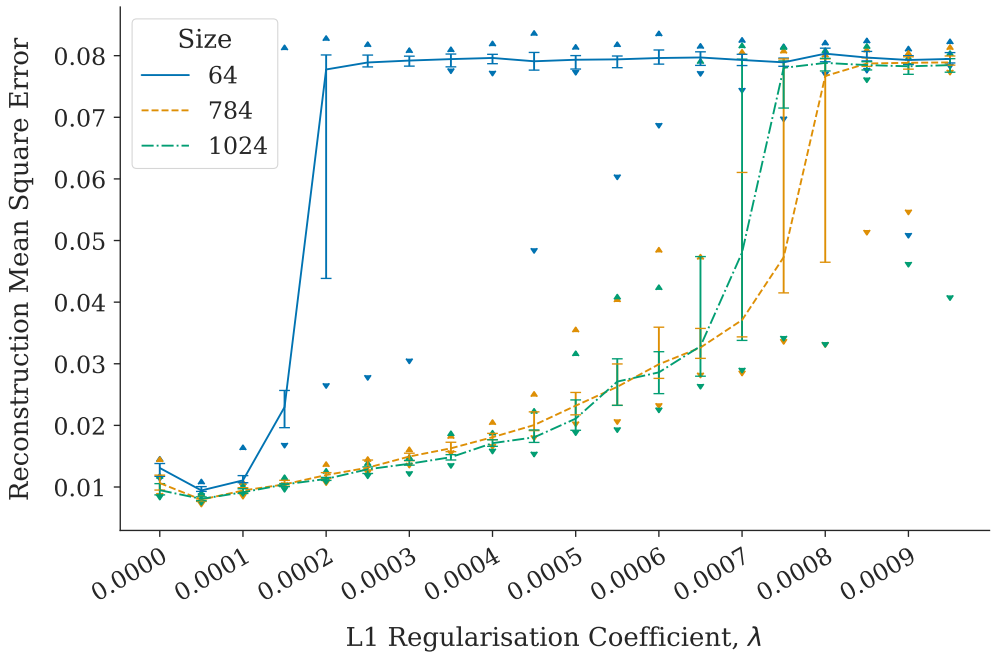
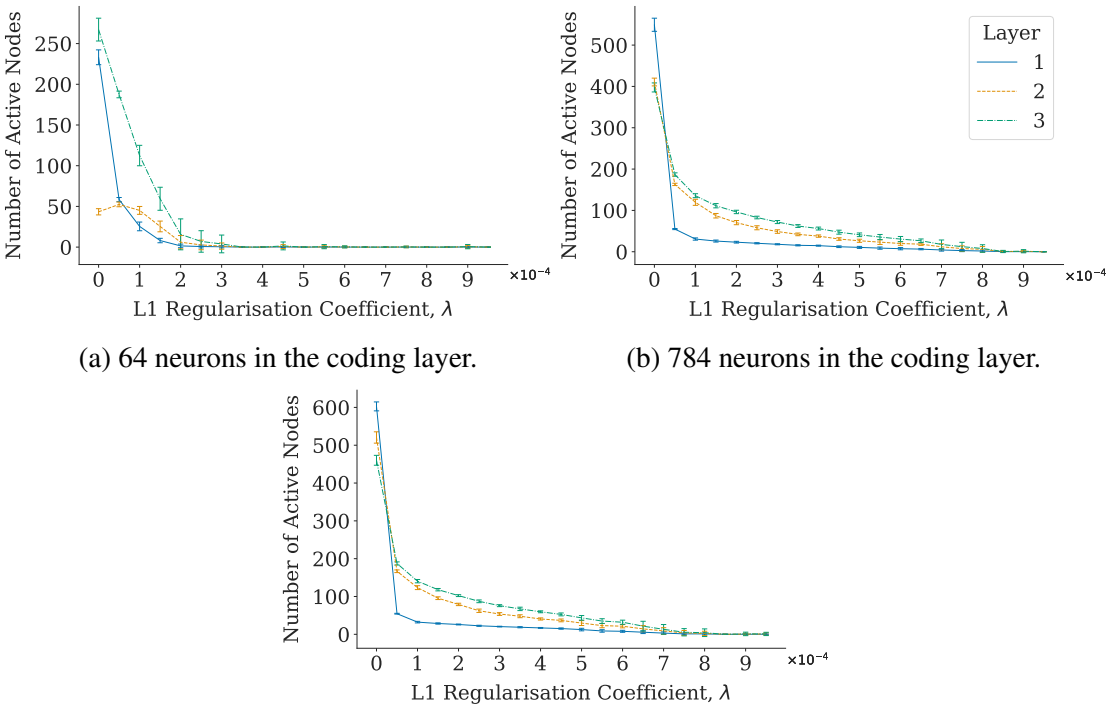


Figure 4.14: Median mean square error between the input and reconstructed image for autoencoders in which *all* layers were regularised. These autoencoders have three hidden layers - one encoding layer, one decoding layer, and one central "code" layer. Error bars indicate the interquartile range, upward facing triangles show the maximum recorded MSE, and downward facing triangles show the minimum recorded MSE.

Figure 4.14 shows the median reconstruction mean square error for the three architectures of autoencoder tested (lower is better). As λ was increased, reconstruction MSE also gradually increased. The autoencoder containing 64 nodes in the central layer was much more sensitive to regularisation than the larger autoencoders, with the reconstruction MSE dramatically increasing after $\lambda = 0.0001$. At $\lambda = 0.00005$ there was a small decrease in MSE for all networks, suggesting that for very small values of λ activity regularisation may actually benefit autoencoder performance. As was the case with fully connected networks, the results of training became much more variable as λ increased.

Figure 4.15 shows the number of active nodes in each layer of the three sizes of autoencoder. As λ increased, the number of active nodes per image decreased as expected. Much like the fully connected networks in Chapter 3, the first hidden layer in each network appeared to be more sparse than the following layers. For the network in Figure 4.15a, the number of active nodes in the coding layer actually increased for $\lambda = 0.5 \times 10^{-4}$ before decreasing again. The networks with both 784 and 1024 nodes in the coding layer showed very similar levels of activity for all values of $\lambda > 0$.



(a) 64 neurons in the coding layer.

(b) 784 neurons in the coding layer.

(c) 1024 neurons in the coding layer.

Figure 4.15: Average number of active nodes in each hidden layer of autoencoders in which all layers were regularised, with (a) 64 nodes, (b) 784 nodes, and (c) 1024 nodes in the coding layer. Values were calculated by measuring the average number of nodes active per image for each network, and averaging over the 20 models trained for each test condition. Error bars indicate standard deviation.

Figure 4.15 shows that it is possible to limit activity in an autoencoder while maintaining reasonable performance, as was the case with previously tested architectures. These results also match well with those found by Glorot et al. (2011) [40]. In their paper they trained a series of stacked denoising autoencoders, enforcing sparsity using ReLU activation functions and an ℓ_1 activity penalty. Denoising autoencoders are trained to reconstruct clean data from a noisy input. This process prevents the autoencoder from simple learning an identity function and increases generalisability. "Stacking" refers to a greedy layer-wise pre-training process in which new layers are stacked onto the existing network and pre-trained using the output of the previous layer as the input. The entire network is then fine-tuned as a single network using more training data. They found that reconstruction error was not affected until approximately 85% of nodes are inactive. Figure 4.16 shows a similar result, where the reconstruction MSE was approximately the same until around 85% of nodes become inactive, at which point MSE increased rapidly.

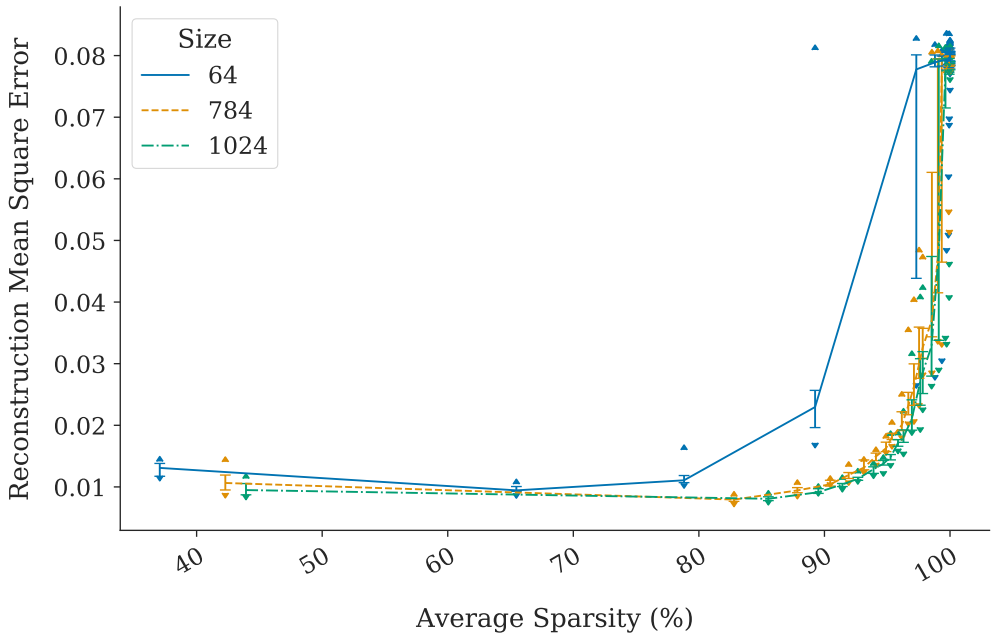
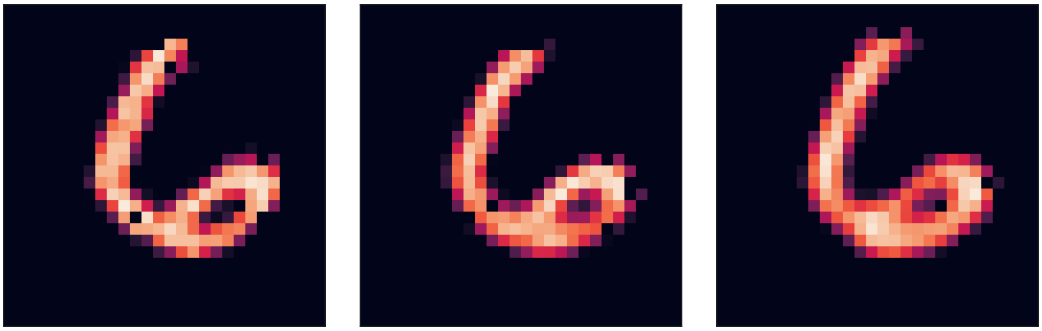


Figure 4.16: Median reconstruction mean square error for autoencoders in which all layers were regularised, for various levels of sparsity. These autoencoders have three hidden layers - one encoding layer, one decoding layer, and one central "code" layer. Average sparsity indicates the percentage of nodes in the network which are inactive for a given input. Error bars indicate the interquartile range, upward facing triangles show the maximum recorded MSE, and downward facing triangles show the minimum recorded MSE.

Figure 4.17 shows some examples of a digit "6" from the MNIST dataset, reconstructed by autoencoders trained with various values of λ. The unregularised overcomplete autoencoder with 1024 nodes in the coding layer performed the best, with an MSE of

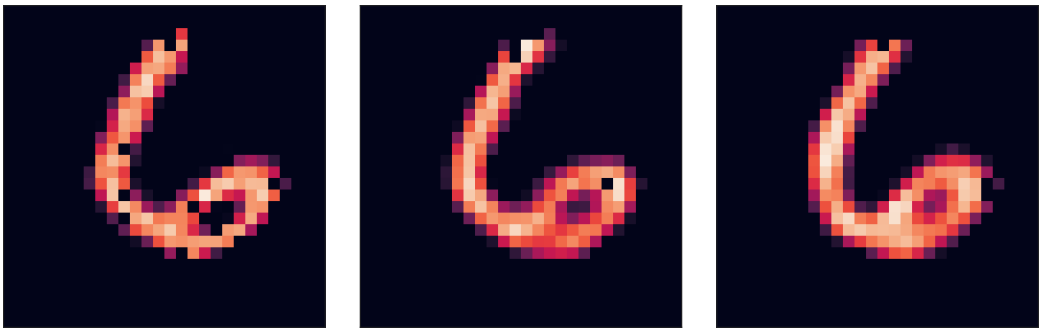
0.007. This, however, is a somewhat trivial case. Since this network is essentially learning the identity function for a set of data, it should be expected to perform well if it has an abundance of neurons to work with. In these particular networks, the unregularised autoencoder using the traditional bottleneck architecture appears to perform the best after the overcomplete autoencoder. Note however that this is one sample out of many autoencoders, and Figure 4.14 suggests that sparse autoencoders may still outperform unregularised autoencoders. To properly test this however, autoencoders should be trained on larger datasets for a longer period of time, and specific architectures should be carefully selected so as to get the best performance possible.



(a) $\lambda = 0$
$MSE = 0.007$
499 of 1024 coding neurons active

(b) $\lambda = 5 \times 10^{-4}$
$MSE = 0.014$
176 of 1024 coding neurons active

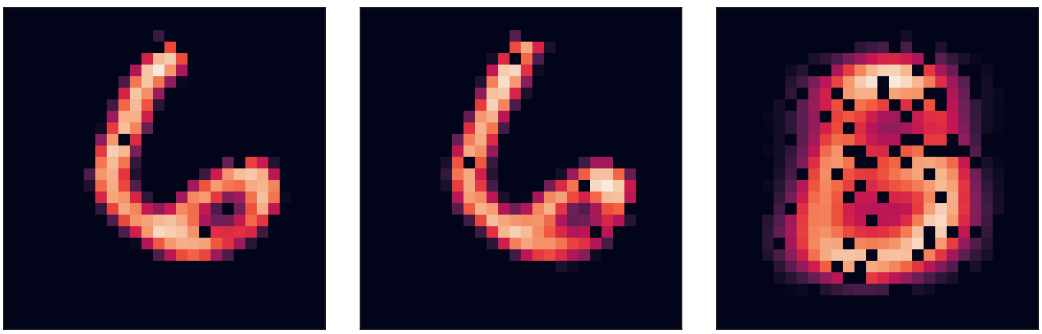
(c) $\lambda = 0.0002$
$MSE = 0.012$
83 of 1024 coding neurons active



(d) $\lambda = 0$
$MSE = 0.012$
421 of 784 coding neurons active

(e) $\lambda = 5 \times 10^{-4}$
$MSE = 0.012$
155 of 784 coding neurons active

(f) $\lambda = 0.0002$
$MSE = 0.014$
78 of 784 coding neurons active



(g) $\lambda = 0$
$MSE = 0.010$
46 of 64 coding neurons active

(h) $\lambda = 5 \times 10^{-4}$
$MSE = 0.013$
52 of 64 coding neurons active

(i) $\lambda = 0.0002$
$MSE = 0.073$
7 of 64 coding neurons active

Figure 4.17: Examples reconstruction outputs from each autoencoder architecture tested, for various values of λ. The input was the digit "6", and the output is the autoencoders attempt to reconstruct the input.

CHAPTER 5

Discussion

In Chapter 3 the results of regularising fully connected networks by the activity of their nodes were presented, and a few main conclusions were drawn. First and foremost it was found that, given certain conditions, sparse activity can be created in a network with little to no hit to performance. This is important as it shows that sparsity is a viable principle of information processing and encoding outside the context of linear coding models. As discussed in Chapter 1, a dense population code in the visual system would be incredibly inefficient, and a local code would not have sufficient capacity to represent the true complexity of an organism's environment. It is important then that sparsity is a viable principle of information coding at a fundamental level – that it is viable in both linear coding models and fully connected neural networks then is encouraging.

It is currently unclear how useful sparse activity in fully connected networks is. In Chapter 3 it was suggested that activity regularisation may be a useful technique to prevent overfitting, and that it could be used to aid in model selection or architecture optimisation. However, more work is required to determine whether activity regularisation can indeed be used for these purposes, and this could be a direction for future work. It would also be useful to further explore how sparse fully connected networks behave with other types of data.

It was also shown that the ReLU activation function plays a key role in creating ℓ_0 sparsity. A network in which the activation function is not equal to zero anywhere, like sigmoid, cannot achieve sparsity by the ℓ_0 measure. They can, however, be considered sparse by "soft" measures such as the ℓ_1 measure or the Gini index. While it has been argued that "hard" sparsity produces codes more like those found in the brain [7], and that ReLU neurons may be more biologically realistic than sigmoid and tanh neurons [40], soft sparsity should not be entirely discounted. Cortical neurons spontaneously activate and have a non-zero firing rate (i.e., activations per second) even in the dark [46, 47]. If the activation value of a node in a neural network represents the firing rate of a neuron, then perhaps soft sparsity may not be entirely biologically unrealistic. It also appears that soft sparse constraints such as the ℓ_1 norm, in combination with the ReLU activation function, can approximate hard sparsity and cause a reduction in the

number of active nodes in a network.

Chapter 3 also demonstrated how sparsity varies across layers in a deep network. Lower levels of sparsity were seen in the first hidden layer, while subsequent layers had greater levels of sparsity. This pattern was repeated in both convolutional networks and autoencoders in Chapter 4. Section 3.2.3 argued that this was largely a result of the back-propagation equations for the regularisation term. While the way in which sparsity is created in the neural networks presented in this thesis may weaken the connection of these models to the brain, it does show that sparse and *useful* activity can be generated via processes other than the lateral inhibition/competition between neurons that was seen in the linear coding models. Activity regularisation tended to shift the distribution of weights to be negatively skewed. As a result the input to a node, $z_i = \underset{\sim}{\boldsymbol{w}}_i \underset{\sim}{\boldsymbol{a}} + b_i$, was more likely to be less than zero, and nodes were less likely to activate overall. This could be considered as *inter-layer* inhibition (as opposed to lateral inhibition between neurons within a layer), resulting in a situation where only the nodes which represent the most important or prominent features of an input have the necessary inputs to become active. However, node biases add a layer of complexity to this interpretation. Note also that in neural networks, once training is complete the weights and biases are set, and sparsity is then created in the *feed-forward* process where information flows through the network from input to output. In comparison, once one of the linear coding models presented in Chapter 1 has learnt a dictionary of basis functions, the optimal set of coefficients are learnt for every image individually. Hence in linear coding models, sparsity is created in the *learning* process.

It is also unclear whether the distribution of weights was similar for each neuron, or if some neurons had far more negative weights than others, as the weights leading into each individual neuron was not investigated. ReLU neurons with a large number of negative weights would be more likely to die in a deep network, becoming inactive for all inputs. If it is the case that only some neurons have many negative weights, then the negative skew in the distribution of weights would not be representative of the effect of regularisation on the weights of the entire network. Instead, this skew would simply represent the degree to which ReLU neurons are dying in such networks.

Chapter 4 reproduced many of these results in convolutional networks and autoencoders, demonstrating that sparse and useful activity can be created in a variety of architectures. Sparse convolutional networks were able to learn features comprised of far fewer components without a significant hit to performance, reducing the amount of redundant information represented in each feature. As in fully connected networks, activity regularisation in CNNs had the effect of negatively skewing the distribution of weights. Structures and patterns that were present in kernels of unregularised networks also appeared to be more localised in regularised networks. This again may have the effect that only the most prominent or significant features would be extracted. In other words, activity regularisation could have the effect of increasing the *specificity* of the network in terms of feature extraction. While there may be many features of an input that could aid in its classification, it is possible to identify an input from only a few key features

that may be unique to, or more prominent in, a given class.

In autoencoders, activity regularisation allowed low-dimensional representations of data to be learnt using overcomplete autoencoders. However, the properties of the code (i.e., the pattern of activations in the coding layer) learnt by regularised and unregularised autoencoders was not quantitatively compared, and so it is unclear whether overcomplete autoencoders can be used to learn representations with different properties to those with the traditional bottleneck architecture. The relationship between sparsity in autoencoders and their general performance matched well with results found by Glorot et al. [40], with performance not being affected until around 85% of nodes in the network became inactive.

5.1 Conclusion

Deep learning is a vast and versatile field of machine learning. Neural networks initially present as an alternative computational avenue in which to investigate sparse coding in the brain. They allow the prospect of exploring sparse coding in hierarchical structures – something which the majority of previous models do not do. However the complexity of deep learning models presents its own challenges, and the simple implementations presented in this thesis cannot represent the full range of biological mechanisms that might induce sparsity. Linear coding models, while not hierarchical, allow inhibiting connections between neurons within a layer, as well as self inhibition, to induce sparsity. The neural networks used in this thesis, however, are strictly feed forward and only allow nodes to inhibit those in the following layer. As such, the competition and inhibition present in previous models does not occur. Additionally, the way in which sparsity changes across the layers in a deep network is intimately connected to the way in which the network learns. Because of this, it may not be possible to make any deductions about the role of sparsity in the brain and its importance across the different layers of the visual system. This work does show, however, that it is indeed possible to produce sparse but useful activity in a range of neural network architectures without a signifiant loss in performance. This supports the idea that sparsity can be a useful coding principle in a range of contexts.

Sparsity also may have its uses in deep learning. Sparsely activating CNNs could be used to extract features from data in which redundant information is removed, and activity regularisation can be used to allow overcomplete autoencoders to learn low-dimensional representations of data. More work is needed however to better characterise the way in which sparse networks differ to unregularised networks. Future work should include quantitative analyses of the receptive fields of nodes, as well as the differences between the features and kernels in regularised and unregularised CNNs.

Another option for future work is modifying fully connected networks so that biases are strictly negative as described in Section 3.2.3. Additionally, these networks could be

expanded to incorporate components from other modern neural network architectures in order to form more biologically realistic models.

Ultimately, sparse coding is a useful coding principle, and is viable in a range of contexts including linear codes and deep neural networks. It is clearly an important principle not only in the visual cortex of the brain, but in other areas as well. It is still unclear whether sparsity is a fundamental guiding principle in the brain, or whether it emerges from some deeper principles, and further investigations into the benefits and the role of sparsity in the visual system and neural networks, as well as the mechanisms by which it can be generated in a variety of contexts, should continue in future work.

APPENDIX A

Additional Figures

A.1 Activation Value Distributions

Figures A.1 and A.2 show examples of the distribution of activities in single layer ReLU and sigmoid fully connected networks with 64 hidden layer nodes, for a range of λ values.

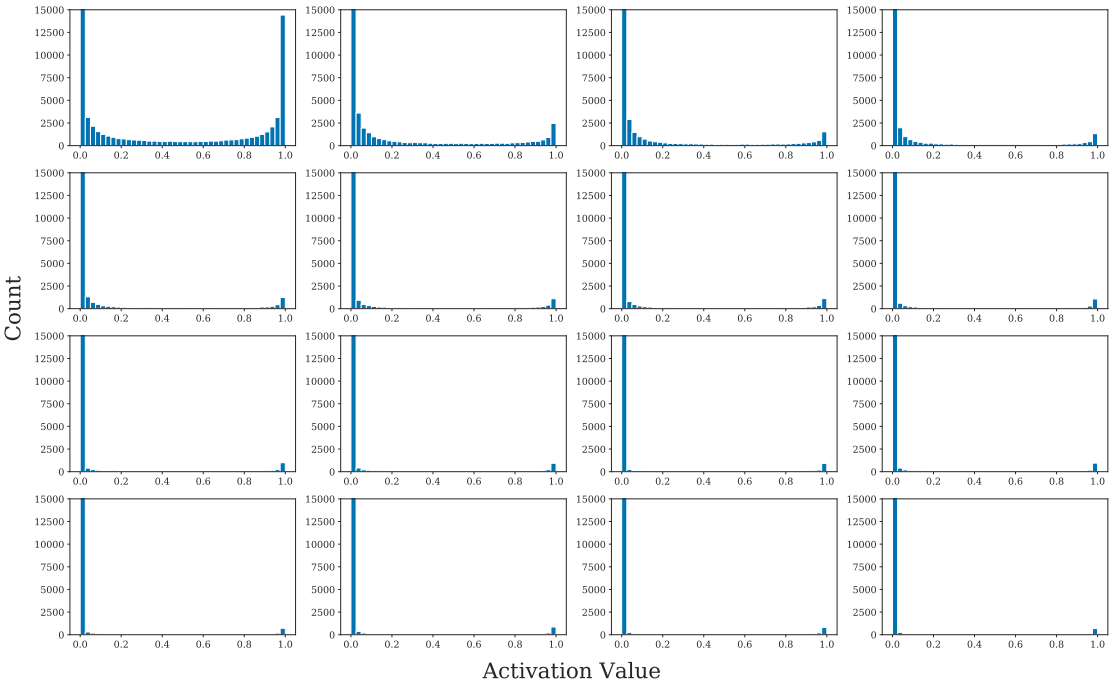


Figure A.2: Distribution of activation values for 1000 images from a 64 node single layer fully connected sigmoid network, for a range of λ values. λ ranges from 0 to 0.12 in steps of 0.008, from left to right, top to bottom ($\lambda = 0$ for top left, $\lambda = 0.12$ for bottom right).

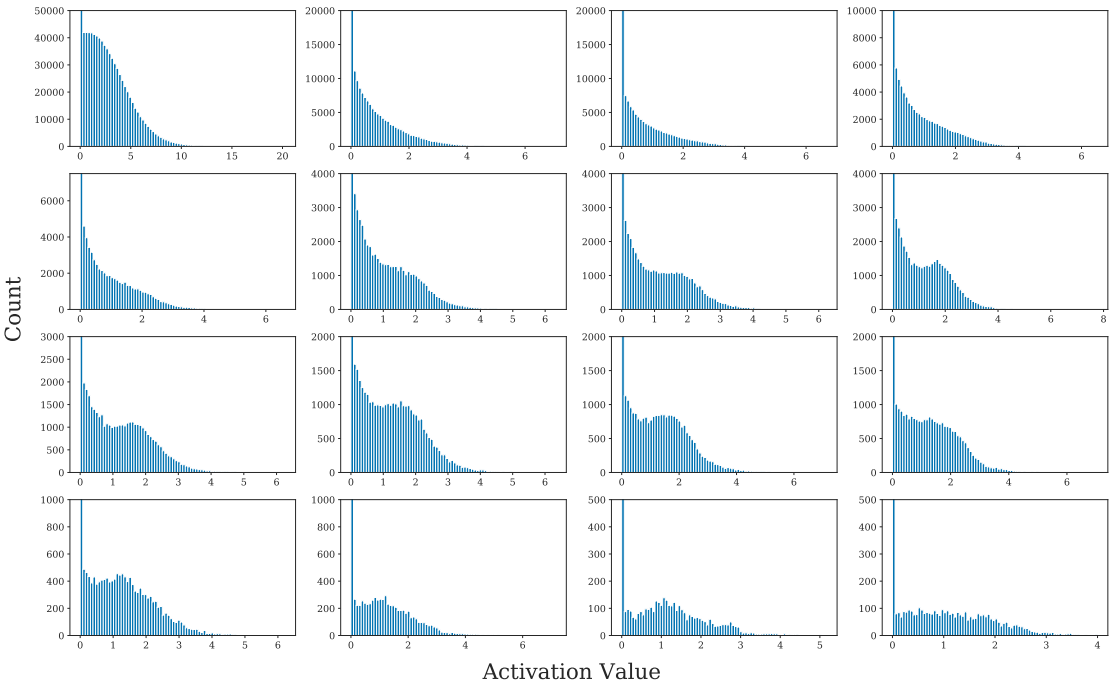
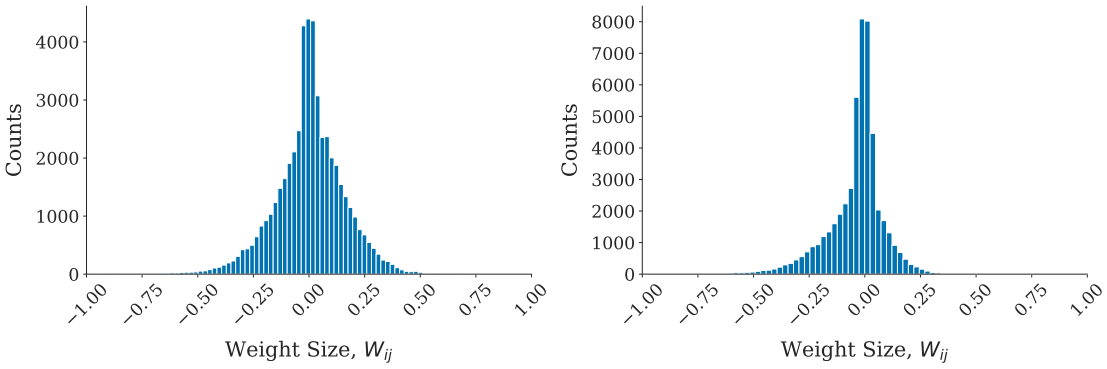


Figure A.1: Distribution of activation values for 1000 images from a 64 node single layer fully connected ReLU network, for a range of λ values. λ ranges from 0 to 0.12 in steps of 0.008, from left to right, top to bottom ($\lambda = 0$ for top left, $\lambda = 0.12$ for bottom right).

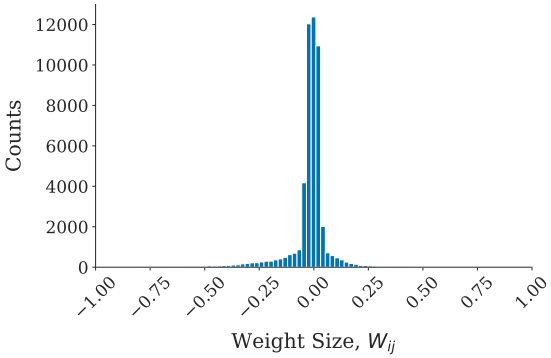
A.2 Weight Distributions

Figures A.3 and A.4 show examples of the distribution of weights in single layer ReLU and sigmoid fully connected networks with 64 hidden layer nodes.

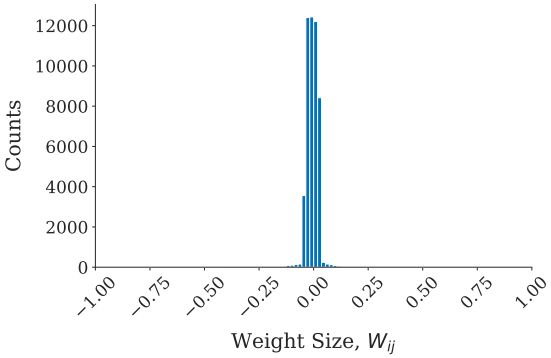


(a) $\lambda = 0$

(b) $\lambda = 0.016$

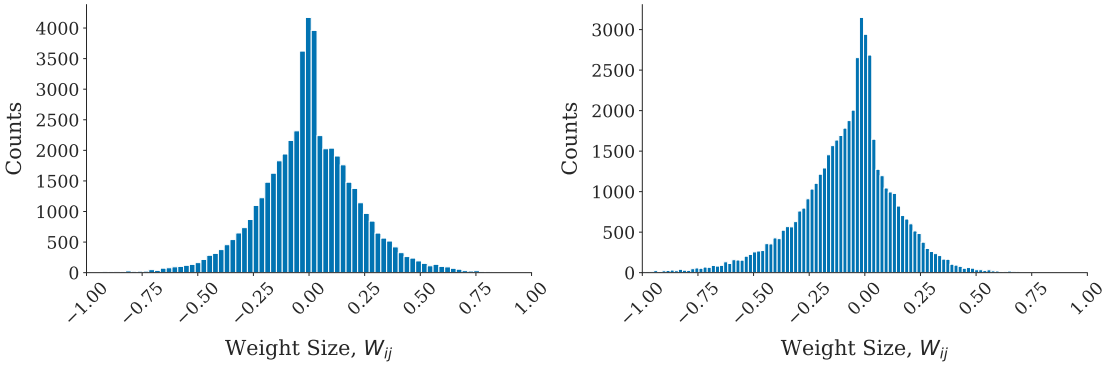


(c) $\lambda = 0.04$



(d) $\lambda = 0.08$

Figure A.3: Distribution of weights from all layers in a single layer networks with a ReLU activation function, for varying λ values. These particular weights are from a model with 64 units in the hidden layer.



(a) $\lambda = 0$ (b) $\lambda = 0.016$

(c) $\lambda = 0.04$ (d) $\lambda = 0.08$

Figure A.4: Distribution of weights from all layers in a single layer networks with a sigmoid activation function, for varying λ values. These particular weights are from a model with 64 units in the hidden layer.

APPENDIX B

Selected Proofs and Derivations

B.1 Derivation of the Backpropagation Equations for Fully Connected Networks

The following derivation was adapted from *Notes on Backpropagation* [48] and notes for *The Mathematical Engineering of Deep Learning* [36]. In particular, the derivation of B.9 was adapted from these notes. The final representation of Equations B.13 and B.14 and the remaining steps to derive them are my own.

Consider the softmax equation:

$$\sigma(\underset{\sim}{\boldsymbol{z}})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \qquad \text{for } i = 1, \ldots, K \text{ and } \underset{\sim}{\boldsymbol{z}} \in \mathbb{R}^K \tag{B.1}$$

And the cross entropy loss for a multi-class prediction:

$$C(y_i, t_i) = -\sum_{i=1}^{K} t_i \log(y_i) = -\log \prod_{i=1}^{K} (y_i)^{t_i} \tag{B.2}$$

We wish to find the partial derivatives of the loss function with respect to each weight, $\frac{\partial C}{\partial \mathbf{W}_{ij}^{(l)}}$. Recall that the state of the nodes in layer l (before applying the activation function) is $\underset{\sim}{\boldsymbol{z}}^{(l)} = \mathbf{W}^{(l)} \underset{\sim}{\boldsymbol{a}}^{(l-1)} + \underset{\sim}{\boldsymbol{b}}^{(l)}$. So by the chain rule, the derivatives of the weights in the l^{th} layer are

$$\frac{\partial C}{\partial W_{ij}^{(l)}} = \frac{\partial C}{\partial z_i^{(l)}} a_j^{(l-1)} \tag{B.3}$$

$$\frac{\partial C}{\partial b_i^{(l)}} = \frac{\partial C}{\partial z_i^{(l)}} \tag{B.4}$$

Note that $\dfrac{\partial C}{\partial z_i^{(l)}}$ must be found via the chain rule. However, knowing that the output layer uses the softmax activation function, this process can be simplified by finding the partial derivatives of the weights in the first layer.

Let the output layer be the L^{th} layer. The outputs of this layer are $y_i = \sigma(\underset{\sim}{\boldsymbol{z}}^{(L)})_i$, where $\underset{\sim}{\boldsymbol{z}}^{(L)} = \mathbf{W}^{(L)}\underset{\sim}{\boldsymbol{a}}^{(L-1)} + \underset{\sim}{\boldsymbol{b}}^{(L)}$. Then the partial derivatives for the weights in the final layer are:

$$\frac{\partial C}{\partial W_{ij}^{(L)}} = \frac{\partial C}{\partial z_i}\frac{\partial z_i}{\partial W_{ij}^{(L)}}. \tag{B.5}$$

Now, since every output y_i is a function of the internal state of every output node, z_i, then the derivative with respect to z_i must take into account every output y_k for $k = 1, \ldots, K$, so that

$$\frac{\partial C}{\partial z_i} = \sum_{k=1}^{K}\frac{\partial C}{\partial y_k}\frac{\partial y_k}{\partial z_i}. \tag{B.6}$$

Now, considering the first term in Equation B.6 we get

$$\begin{aligned}\frac{\partial C}{\partial y_k} &= \frac{\partial}{\partial y_g}\left(-\sum_{g=1}^{K} t_g \log(y_g)\right)\\ &= -\frac{t_k}{y_k},\end{aligned} \tag{B.7}$$

and for the second term, when $k \neq i$

$$\begin{aligned}\frac{\partial y_k}{\partial z_i^{(L)}} &= \frac{\partial}{\partial z_i^{(L)}}\left(\frac{e^{z_k}}{\sum_{i=1}^{K} e^{z_i}}\right)\\ &= \frac{-e^{z_k}}{\left(\sum_{i=1}^{K} e^{z_i}\right)^2}\cdot e^{z_i}\\ &= -y_k y_i,\end{aligned}$$

amd when $k = i$

$$\begin{aligned}\frac{\partial y_k}{\partial z_i^{(L)}} &= \frac{\partial}{\partial z_k^{(L)}}\left(\frac{e^{z_k}}{\sum_{i=1}^{K} e^{z_i}}\right)\\ &= \frac{e^{z_k}}{\sum_{i=1}^{K} e^{z_i}} - \frac{(e^{z_k})^2}{\left(\sum_{i=1}^{K} e^{z_i}\right)^2}\\ &= y_k(1 - y_k)\\ &= y_i(1 - y_i)\end{aligned}$$

Hence the second term in Equation B.6 becomes

$$\frac{\partial y_k}{\partial z_i^{(L)}} = \begin{cases} -y_k y_i, \text{ for } k \neq i \\ y_i(1-y_i), \text{ for } k = i \end{cases} \tag{B.8}$$

As a result, the partial derivative of the cost function with respect to the final layer internal states is

$$\begin{aligned} \frac{\partial C}{\partial z_i^{(L)}} &= \sum_{k=1}^{K} \frac{\partial C}{\partial y_k} \frac{\partial y_k}{\partial z_i} \\ &= -\sum_{k=1}^{K} \frac{t_k}{y_k} \frac{\partial y_k}{\partial z_i} \\ &= -t_i(1-y_i) + \sum_{k \neq i}^{K} t_k y_i \\ &= -t_i + y_i \sum_{k=1}^{K} t_k \\ &= y_i - t_i \end{aligned} \tag{B.9}$$

Finally, the second term in Equation B.5 is

$$\begin{aligned} \frac{\partial z_i^{(L)}}{\partial W_{ij}^{(L)}} &= \frac{\partial}{\partial W_{ij}^{(L)}} (\underset{\sim}{\boldsymbol{w}}_i^{(L)} \underset{\sim}{\boldsymbol{a}}^{(L-1)} + b_i^{(L)}) \\ &= a_j^{(L-1)}. \end{aligned}$$

Therefore the partial derivatives of the weights in the last layer (layer $l = L$) are given as

$$\frac{\partial C}{\partial W_{ij}^{(L)}} = (y_i - t_i) a_j^{(L-1)}. \tag{B.10}$$

For the weights in layers $l < L$, the backpropagation equations become

$$\frac{\partial C}{\partial W_{ij}^{(l)}} = \sum_{n}^{N_L} \frac{\partial C}{\partial z_n^{(L)}} \frac{\partial z_n^{(L)}}{\partial a_i^{(l)}} \frac{\partial a_i^{(l)}}{\partial z_i^{(l)}} a_j^{(l-1)} \tag{B.11}$$

Now, $\dfrac{\partial a_i^{(l)}}{\partial z_i^{(l)}}$ is the derivative of the activation function in layer l. In the case where the rectified linear unit (ReLU) activation function is used, this becomes

$$\frac{\partial a_i^{(l)}}{\partial z_i^{(l)}} = ReLU'(z_i^{(l)}) = \begin{cases} 1, \text{ if } z_i^{(l)} > 0 \\ \text{undefined if } z_i^{(l)} = 0 \\ 0, \text{ if } z_i^{(l)} < 0 \end{cases} \tag{B.12}$$

Hence for a multi-class fully connected network using the ReLU activation function in hidden layers, the softmax function in the final layer, and the Cross Entropy cost function, the backpropagation equations for the weights are

$$\frac{\partial C}{\partial W_{ij}^{(l)}} = \begin{cases} \sum_{n}^{N_L} \frac{\partial z_n^{(L)}}{\partial a_i^{(l)}} (y_n - t_n) a_j^{(l-1)}, \text{ if } z_i^{(l)} > 0 \\ 0 \text{ Otherwise} \end{cases} \tag{B.13}$$

and for the biases, the partial derivatives are

$$\frac{\partial C}{\partial b_i^{(l)}} = \begin{cases} \sum_{n}^{N_L} \frac{\partial z_n^{(L)}}{\partial a_i^{(l)}} (y_n - t_n), \text{ if } z_i^{(l)} > 0 \\ 0 \text{ Otherwise} \end{cases} \tag{B.14}$$

The sum over n is due to the fact that all $z_n^{(l)}$ in the final layer are functions of all weights in the previous layers.

APPENDIX C

Code Examples

This section presents examples of the code used in this thesis. In particular, it demonstrates model creation in *PyTorch*, the training function used to train and regularise networks, functions used to measure average accuracy and sparsity on test data, and an example of a typical script used to train a set of networks. In many cases some changes need to be made to adapt each example to specific cases (for example, changes must be made to the training function in order to train autoencoders using the Mean Square Error cost function, as the training function shown here assumes the Cross Entropy cost function is used).

Green text indicates comments, purple text indicates strings, and blue text represents other language-specific functions.

C.1 Creating Models in PyTorch

There are a number of methods for defining the structure of a network in PyTorch, but two main components are always required. The components of the model (i.e., the type and number of layers) are defined in the *initialisation* method (`__init__`). The way in which information passes through the components of the network is then defined in the *forward* method.

C.1.1 Fully Connected Networks

Here, a fully connected network with two hidden layers, and 128 neurons in each hidden layer, is defined. The two hidden layers, as well as the output layer, are defined in the `__init__` method. Each layer defined in this method may be referred to as a *module*. `nn.Linear()` modules are fully connected - i.e. every input to the layer is connected to every node in the layer.

Note that a module called `self.relu()` is also defined in the `__init__` method. This is because forward hooks (the method used to capture activation values, described in Appendix C.2.2) capture the output of whichever *modules* defined in `__init__` that they are registered to. By instead implementing the ReLU activation function as a single method, then any time this method is called its outputs will be captured.

The ReLU activation function module is then implemented in the *forward* method of the network, so that the output of each layer passes through the ReLU function before being passed to the next layer.

```
class FC2_net(nn.Module):
    def __init__(self):
        super(FC2_net, self).__init__()
        self.fc1 = nn.Linear(784, 128)
        self.fc2 = nn.Linear(128, 128)
        self.out = nn.Linear(128, 10)     #output layer

        #Instantiate Relu so hook can be registered
        #nn.Relu() is for modular defn, i.e. to be used in sequential model
        #nn.functional.relu is functional version - if using in forward method
        self.relu = nn.ReLU()

    #Forward equations
    def forward(self, input):
        x = input.reshape(input.shape[0], -1)
        x = self.relu(self.fc1(x))
        x = self.relu(self.fc2(x))
        output = self.out(x)    #CLE loss fn has softmax build in, so leave outputs as raw
        return output
```

C.1.2 Convolutional Networks

The convolutional network below is constructed in a similar manner to the fully connected network, however the ReLU activation function is wrapped into each layer module using *PyTorch's* `nn.Sequential()` method. This method combines multiple modules into a single module.

```
def GenerateConv2(input_size: tuple, kernel_size = 3):
    """
    """
    class Conv2(nn.Module):
        def __init__(self, input_size, kernel_size):
            super(Conv2, self).__init__()

      ### Parameters ###
            self._input_size = input_size
            self.kernel_size = kernel_size
            self._input_channels = input_size[0]
            self.input_len = input_size[1]
            self.C1_outsize = floor(OutSize(self.input_len, self.kernel_size, 0, 1))
            self.C2_outsize = floor(OutSize(self.C1_outsize, self.kernel_size, 0, 1))

      ### Layers ###
            self.C1 = nn.Sequential(
                nn.Conv2d(self._input_channels, 6, kernel_size=self.kernel_size, stride=1, padding=0),
```

```
                nn.ReLU()
            )

            self.C2 = nn.Sequential(
              nn.Conv2d(6, 16, kernel_size=self.kernel_size, stride=1, padding=0),
              nn.ReLU()
            )

            self.FC1 = nn.Sequential(
                nn.Linear(16 * self.C2_outsize ** 2, 64),
                nn.ReLU()
            )
            self.FC2 = nn.Sequential(
                nn.Linear(64, 10)
            )

      #self.relu = nn.ReLU()

        def forward(self, x):
            out = self.C1(x)
            out = self.C2(out)
            out = out.reshape(out.shape[0], -1)
            out = self.FC1(out)
            out = self.FC2(out)
            return out

    return Conv2(input_size, kernel_size)
```

```
def OutSize(input_size, kernel_size, padding, stride):

    return math.floor( (input_size + 2 * padding - kernel_size) / (stride) + 1)
```

C.1.3 Autoencoders

```
def AutoEnc_2(layer_sizes = (784, 400, 64, 400, 784), **kwargs):
    class AutoEncoder(nn.Module):
        def __init__(self, layer_sizes):
            super().__init__()

      #Parameters
            self.input_len = layer_sizes[0]
            self.e1_size = layer_sizes[1]
            self.code_size = layer_sizes[2]
            self.d1_size = layer_sizes[3]

      #Layers
            self.encoder_h1 = nn.Sequential(
                nn.Linear(self.input_len, self.e1_size),
                nn.ReLU()
                )
            self.code = nn.Sequential(
                nn.Linear(self.e1_size, self.code_size),
                nn.ReLU()
              )
            self.decoder_h1 = nn.Sequential(
                nn.Linear(self.code_size, self.d1_size),
                nn.ReLU()
                )
            self.decoder_out = nn.Sequential(
                nn.Linear(self.d1_size, self.input_len),
                nn.ReLU()
```

```
28                )
29
30          #Forward equations
31          def forward(self, input):
32              x = input.reshape(input.shape[0], -1)
33              x = self.encoder_h1(x)
34              code = self.code(x)
35              x2 = self.decoder_h1(code)
36              reconstruction = self.decoder_out(x2)
37
38              return reconstruction
39
40      return AutoEncoder(layer_sizes, **kwargs)
```

C.2 Training and Regularisation

The primary purpose of the following training function is to train fully connected neural networks with an activity penalty in order to create a sparsely activating network. On top of training, measurements of accuracy and loss for both the training and validation datasets are also recorded.

Activity regularisation is implemented in lines $69-77$. This simply involves calculating the average ℓ_1 norm of the activations of the network per image, and adding this to the loss which is initially calculated in line 57.

C.2.1 Training Function

```
1   def L1_train(model, dataloaders, loss_criterion, optimizer, hook, num_epochs = 5,
        L1_lambda = 0.0001, print_progress = True, norm_ord = 1, _num_registered_modules =
         None):
2       """Trains a fully connected network with Lp regularisation on the activations of
        nodes in hidden layers, and
3       tracks training and validation accuracy and loss.
4
5       Args:
6           - model: Pytorch network.
7           - dataloaders (dict): Dictionary of dataloaders of the form {'train':
        trainloader, 'val': validationloader}.
8           - loss_criterion (Class): Loss function (e.g. Cross Entropy).
9           - optimizer (Class): Gradient descent optimizer.
10          - hook (Class): Forward hook to capture layer outputs (e.g. OutputHook() class)
        . Hook must be registered to the desired modules before training.
11          - num_epochs (int, optional): Number of training epochs. Defaults to 5.
12          - L1_lambda (float, optional): L1 regularisation coefficient. Defaults to
        0.0001.
13          - print_progress (bool, optional): Whether or not training progress should be
        printed each epoch. Defaults to True.
14          - norm_ord (int): Order of Lp regularisation, i.e. L1, L2, etc.
15
16      Returns:
17          - Training loss (List): Average loss over each epoch for training data.
18          - Training accuracy (List): Percentage of correct predictions over the last
        epoch for training data.
19          - Validation loss (List): Average loss over each epoch for the validation data.
```

```
20          - Validation accuracy (List): Percentage of correct predictions for validation
    data after each epoch.
21          - L1 Loss (List): Average L1 cost over each epoch for the training data
22      """
23      since = time()
24      train_loss_list = []
25      train_accuracy_list = []
26      val_loss_list = []
27      val_accuracy_list = []
28      L1_loss = []
29
30      batch_size = dataloaders['train'].batch_size    #batch size for training data
31      no_batches = {'train': len(dataloaders['train'].dataset)/batch_size, 'val': len(
    dataloaders['val'].dataset)/batch_size}   #Number of batches in each set
32
33      #Begin epoch
34      for epoch in range(num_epochs):
35          #Set models to appropriate mode
36          for phase in ['train', 'val']:  #Training and validation phases
37              if phase == 'train':
38                  model.train()
39              else:
40                  model.eval()
41
42              correct = 0
43              running_loss = 0
44              for inputs, labels in dataloaders[phase]: #For each batch of images in the
    dataset
45                  inputs = inputs.to(device)
46                  labels = labels.to(device)
47
48                  optimizer.zero_grad()   #Zero gradients
49
50                  ### Training ###
51                  #If in training phase, track gradients, otherwise do not.
52                  with torch.set_grad_enabled(phase == 'train'):
53
54                      ###Forward pass###
55                      outputs = model(inputs)
56                      #Calculate Mean Cross Entropy loss per input image
57                      loss = loss_criterion(outputs, labels)
58
59                      #Double check that the correct number of activations have been
    hooked.
60                      #Seemed to be an issue with this sometimes, not sure why.
61                      if _num_registered_modules is not None:
62                          _hook_len = len(hook.outputs)
63                          if _hook_len != _num_registered_modules:
64                              raise Exception(f'Number of modules registered does not
    match the number of modules actually hooked: {_num_registered_modules} modules
    registered, {_hook_len} modules hooked.')
65
66                      L1_penalty = 0
67                      running_L1 = 0
68                      #L1 penalty calculation
69                      for activations in hook.outputs:    #For activations in each layer
70                        #Activations is a tensor - we want vector L1 norm, so flatten
    into vector first
71                          activity = torch.flatten(activations)
72                          L1_penalty = torch.linalg.norm(activity, norm_ord) #Vector norm
     of activations
73                          running_L1 += L1_penalty
74
75                      if L1_lambda != 0:
76                      #Add average L1 penalty per image
77                          loss += (L1_lambda/batch_size) * running_L1
```

```
78                      if phase == 'train':
79                          L1_loss.append(running_L1.detach()/batch_size)
80
81                      ###Backward pass###
82                      if phase == 'train':
83                          loss.backward()     #Backpropagation
84                          optimizer.step()    #Update weights
85
86                      hook.clear()        #Clear L1 activity hooks, otherwise they will
    accumulate
87
88                  ###Statistics###
89                  _, predictions = torch.max(outputs, norm_ord)  #Gives index of max in
    each row
90                  running_loss += loss.item()
91                  correct += torch.sum(predictions == labels.data)  #Number of correct
    predictions
92
93              #Epoch loss - loss averaged over entire epoch
94              #epoch_loss is mean loss per image over the epoch. Running loss is already
95              #Batch_no lots of mean loss per image, so divide by Batch_no to get mean
96              #loss per image
97              epoch_loss = running_loss / (no_batches[phase])
98              epoch_accuracy = correct.double() / len(dataloaders[phase].dataset)
99
100             #Save training and validation loss and accuracy
101             if phase == 'train':
102               train_loss_list.append(epoch_loss)
103               train_accuracy_list.append(epoch_accuracy*100)
104             elif phase == 'val':
105               val_loss_list.append(epoch_loss)
106               val_accuracy_list.append(epoch_accuracy*100)
107
108         #Print progess
109         if print_progress==True:
110             print('Epoch [{}/{}], Train Loss: {:.4f}, Train Accuracy: {:.2f}%, Val Loss
    : {:.4f}, Val Accuracy: {:.2f}%'.format(epoch + 1, num_epochs, train_loss_list
    [-1],train_accuracy_list[-1], val_loss_list[-1], val_accuracy_list[-1]))
111
112     time_elapsed = time() - since
113
114     print('Training complete in {:.0f}m {:.0f}s'.format(time_elapsed // 60,
    time_elapsed % 60))
115     print('Best val Acc: {:4f}'.format(max(val_accuracy_list)))
116
117     return train_loss_list, train_accuracy_list, val_loss_list, val_accuracy_list,
    L1_loss
```

C.2.2 Forward Hook

The following class is a *hook*. The role of a hook is to capture the outputs of a module in a network.

```
1   class OutputHook(list):
2       """
3       Forward hook to capture module outputs. e.g. to capture the outputs of ReLU neurons
        .  Register hooks on a module by:
4
5       hook = OutputHook()
6       model.module.register_forward_hook(hook)
7
```

```
8       where module is the name of the desired module. Activations are then found in
        OutputHook.outputs
9       """
10      def __init__(self):
11          self.outputs = []
12
13      def __call__(self, module, input, output):
14          activations = torch.squeeze(output) #Squeeze to remove empty dimension that for
         some reason occurs, otherwise norm calculation is messed up
15          self.outputs.append(activations)
16
17      def clear(self):
18          self.outputs = []
```

To use a hook to capture activations from a particular module, they must first be instantiated, then registered to the module as follows:

```
1 forward_hook = OutputHook()
2 network.module.register_forward_hook(forward_hook)
```

where `network` is the name of network, and `module` is the module being hooked. For the fully connected networks, the ReLU activation function was implemented as a module called `relu`, and so only needs to be hooked once. For a network such as the convolutional network defined in C.1.2, the ReLU function is incorporated into each module, and each module being regularised needs to be registered separately, e.g. as

```
1 forward_hook = OutputHook()
2 network.C1.register_forward_hook(forward_hook)
3 network.C2.register_forward_hook(forward_hook)
4 network.FC1.register_forward_hook(forward_hook)
5 network.FC1.register_forward_hook(forward_hook)
```

This allows greater flexibility if only some layers are to be regularised.

C.3 Testing

The following functions were used to test the networks after training and collect various statistics such as accuracy, ℓ_0 and ℓ_1 sparsity, and the Gini index.

C.3.1 Accuracy, Precision, and Recall

```
1  def TestModel(network, images, labels):
2      """Test a network on a test dataset and calculate the accuracy, recall, precision,
       and confusion matrix.
3      Dataloader batch_size must be the size of the entire test dataset.
4
5      Args:
6          - network: Pytorch network
7          - test_dataloader (DataLoader): Pytorch DataLoader containing the test data
8
9      Returns:
10         - Confusion matrix (Tensor): Confusion matrix of predictions - Row = True label
       , Column = Prediction
```

```
        - Accuracy (float): Percentage of predictions that are correct
        - Precision (Tensor): Proportion of predictions for each class that are correct. = True positives / (True positives + False positives)
        - Recall (Tensor): Proportion of each class correctly identified. = True positives / (True positives + False negatives)
    """
    network.eval()

    output = network(images)

    _, predictions = torch.max(output, 1)  #Gives index of max in each row

    confusion_matrix = ConfusionMatrix(labels, predictions) #Row = Truth, col = prediction
    confusion_matrix.to(device)

    TP = torch.diag(confusion_matrix)   #Number of correct predictions - True positives
    actual_positives = torch.sum(confusion_matrix, dim=1) #Total number of images in each class (i.e.TP + FN)
    positive_predictions = torch.sum(confusion_matrix, dim=0)  #Total number of predictions made of each class (i.e. TP + FP)

    class_precision = torch.div(TP, positive_predictions)  #Proportion of predictions for each class that are correct
    class_recall = torch.div(TP, actual_positives)    #Proportion of each class correctly identified
    accuracy = torch.sum(torch.diag(confusion_matrix)) / len(images) * 100

    return confusion_matrix, accuracy, class_precision, class_recall
```

C.3.2 ℓ_0 and ℓ_1 sparsity

```
def Sparsity(hook):
    """Calculate the average and standard deviation L0 and L1 sparsity of a network, given activations from a hook.
    When test data is given to the network as a batch, activations are hooked as a tensor with rows = images, and columns = nodes.
    Sparsity will be calculated per image, then averaged over all images.

    Args:
        - hook (class): Forward hook registered to capture desired activation values.

    Returns:
        - L0_av (float): Average number of active nodes for a given input
        - L0_std (float):
        - L1_av (float):
        - L1_std (float):
    """
    No_layers = len(hook.outputs)
    L0_av = torch.empty((No_layers))
    L0_std = torch.empty((No_layers))
    L1_av = torch.empty((No_layers))
    L1_std = torch.empty((No_layers))

    for layer, activations in enumerate(hook.outputs):
        no_images = hook.outputs[0].shape[0]    #Number of images - usually 10000
        activations = activations.reshape(no_images,-1)
        #Sparsity per image - norm calculated along dim=1 - i.e. across columns, as columns correspond to nodes in a layer.
        #Gives sparsity per image.
        L0_act_per_im = torch.linalg.norm(activations, ord=0, dim=1).detach()
        L1_act_per_im = torch.linalg.norm(activations, ord=1, dim=1).detach()
```

```
28
29          L0_av[layer] =torch.mean(L0_act_per_im)
30          L0_std[layer] =torch.std(L0_act_per_im)
31          L1_av[layer] =torch.mean(L1_act_per_im)
32          L1_std[layer] =torch.std(L1_act_per_im)
33
34      return L0_av, L0_std, L1_av, L1_std
```

C.3.3 Gini Index

The following code was written by Guest (2017) [49].

```
1  def gini(array):
2      """Calculate the Gini coefficient of a numpy array."""
3
4      # All values are treated equally, arrays must be 1d:
5      array = array.flatten()
6
7      if np.amin(array) < 0:  # Values cannot be negative:
8          array -= np.amin(array)
9
10     array += 0.0000001  # Values cannot be 0
11     array = np.sort(array)     # Values must be sorted
12     index = np.arange(1,array.shape[0]+1) # Index per array element
13     n = array.shape[0]  # Number of array elements
14
15     return ((np.sum((2 * index - n  - 1) * array)) / (n * np.sum(array))) # Gini
       coefficient
```

C.4 Example Implementation

Below is an example implementation of a typical script to train a set of fully connected networks. This script trains 20 networks for every combination of model size (8, 32, 64, 128, 256 hidden layer nodes) and regularisation coefficient.

Note that because all hidden layers in the fully connected network use the ReLU activation function, implemented as a module as described above, only one hook needs to be registered. If instead the layers are defined as they are in the convolutional networks and autoencoders, each layer will need to be registered separately (although the same hook may be used for each).

```
1  ##################
2  ### Libraries ###
3  ##################
4  # Pytorch
5  import torch
6  import torch.nn as nn
7  # Math
8  import numpy as np
9  # Sys
10 from time import time
11 import os
12 # My functions
```

```
13 # Contains training function, hook, and other useful functions
14 import my_funcs_src.L1_reg_fun as reg
15 import my_funcs_src.Models as M  # Contains funtions to instantiate models
16 # Cuda
17 if torch.cuda.is_available():
18     dev = "cuda:0"
19     print(f'CUDA is available, running on {torch.cuda.get_device_name(torch.cuda.
       current_device())}\nnum_workers={num_workers}')
20 else:
21     dev = "cpu"
22 device = torch.device(dev)
23
24 ##########################
25 ### Output Directories ###
26 ##########################
27 folders = [i for i in os.listdir() if not os.path.isfile(i)]
28
29 if 'Training' not in folders:
30     os.mkdir('Training')
31 statsdir = './Training/'
32 if 'Models' not in folders:
33     os.mkdir('Models')
34 modeldir = './Models/'
35
36 data_root = r'./data'
37 dataloaders, dataset = reg.import_mnist(root=data_root) #Convenience function to split
       the MNIST dataset into training, validation, and testing splits, and store the
       dataloaders and datasets for each in a dictionary
38
39 ##################
40 ### PARAMETERS ###
41 ##################
42 epochs = 15
43 model_sizes = [8, 32, 64, 128, 256]
44 lambda_range = np.arange(0, 0.12, 0.008)
45 models_per_test = 20
46 number_of_layers = 2
47 number_of_tests = len(model_sizes)*len(lambda_range)*models_per_test
48
49 ##########################
50 ### Statistics/Metrics ###
51 ##########################
52 #Training loss, accuracy, Validation loss, accuracy, and L1 penalty
53 all_TL = torch.zeros((len(model_sizes), len(lambda_range), models_per_test, epochs))
54 all_TA = torch.zeros((len(model_sizes), len(lambda_range), models_per_test, epochs))
55 all_VL = torch.zeros((len(model_sizes), len(lambda_range), models_per_test, epochs))
56 all_VA = torch.zeros((len(model_sizes), len(lambda_range), models_per_test, epochs))
57 L1loss = torch.zeros((len(model_sizes), len(lambda_range), models_per_test, int
       (50000/100*epochs)))
58
59 #############
60 ### Train ###
61 #############
62 count = 0
63 since = time()
64
65 for s, model_size in enumerate(model_sizes):
66     ### Now step through each value of Lambda ###
67     for l, L1_lambda in enumerate(lambda_range):
68         ### And perform the test for each value 20 times ###
69         for r in range(models_per_test):
70
71       #Instantiate model
72             test_model = M.generate_model_FC2(model_size)
73             test_model.to(device)  # Place model on device (either cpu or gpu)
74
```

```
75        #Establish optimisation method, cost function, and training hooks
76              optimiser = torch.optim.Adam(test_model.parameters(), lr=0.001)  #
      Optimisation algorithm
77              criterion = nn.CrossEntropyLoss()  # Cost function
78              training_hook = reg.OutputHook()  # Forward hook
79              test_model.relu.register_forward_hook(training_hook)  # Register hook to
      relu layers
80
81              ### Training ###
82              #Print progress
83              model_name = f'FC{number_of_layers}_{model_size}hu_L{L1_lambda}_r{r}'
84              print(f"Training model {count}/{number_of_tests}|" + "-"*int(np.floor(count
       *(20/number_of_tests))) +
85                    "_"*(20 - int(np.floor(count *(20/number_of_tests)))) +  f"| \
      nTraining {model_name}\n", end = '\r')
86
87              #Actual training step
88              TL, TA, VL, VA, L1_loss, = reg.L1_train(test_model,
89                                                      dataloaders,
90                                                      criterion,
91                                                      optimiser,
92                                                      training_hook,
93                                                      epochs,
94                                                      L1_lambda=L1_lambda,
95                                                      print_progress=True)
96              count += 1
97              #Record training stats
98              all_TL[s, l, r, :] = torch.tensor(TL)
99              all_TA[s, l, r, :] = torch.tensor(TA)
100             all_VL[s, l, r, :] = torch.tensor(VL)
101             all_VA[s, l, r, :] = torch.tensor(VA)
102             all_L1loss[s, l, r, :] = torch.tensor(L1_loss)
103
104             ### Save models ###
105             #Save each test group in separate folder
106             test_group = f'FC{number_of_layers}_{model_size}hu_L{L1_lambda}'
107
108             groups = os.listdir(modeldir)
109             if test_group not in groups:  # If folder for group doesnt exist, make one
110                 os.mkdir(modeldir + f'{test_group}')
111
112             torch.save(test_model.state_dict(), modeldir + f'{test_group}/{model_name}.
      pth')
113
114         ### Save statistics ###
115         #Save every 20 models
116         training_metrics = {'all_TL': all_TL, 'all_TA': all_TA,'all_VL': all_VL, '
      all_VA': all_VA, 'all_L1loss': all_L1loss}
117         if str(model_size) not in os.listdir(statsdir): #separate folder for each size
118             os.mkdir(statsdir + str(model_size))
119
120         for metric in training_metrics:
121             np.save(statsdir + f'{model_size}/{metric}_slre', training_metrics[metric].
      detach().numpy())
122
123 done = time() - since
124
125 print(f'Tested {count} models in {done/60} minutes.')
```

Bibliography

[1] Sylvia C. Pont and Jan J. Koenderink. “Matching Illumination of Solid Objects”. In: *Perception & Psychophysics* 69.3 (2007), pp. 459–468.

[2] Bruno A. Olshausen and David J. Field. “Sparse Coding with an Overcomplete Basis Set: A Strategy Employed by V1?” In: *Vision Research* 37.23 (1997), pp. 3311–3325.

[3] Bruno A. Olshausen. “Principles of Image Representation in Visual Cortex”. In: *The visual neurosciences* 2 (2003), pp. 1603–1615.

[4] Horace B. Barlow. “Redundancy Reduction Revisited”. In: *Network: Computation in Neural Systems* 12.3 (2001), pp. 241–253.

[5] Dylan M. Paiton. “Analysis and Applications of the Locally Competitive Algorithm”. Thesis. 2019.

[6] Bruno A. Olshausen and David J. Field. “Emergence of Simple-Cell Receptive Field Properties by Learning a Sparse Code for Natural Images”. In: *Nature* 381.6583 (1996), pp. 607–609.

[7] Martin Rehn and Friedrich T. Sommer. “A Network That Uses Few Active Neurones to Code Visual Input Predicts the Diverse Shapes of Cortical Receptive Fields”. In: *Journal of Computational Neuroscience* 22.2 (2007), pp. 135–146.

[8] Adam Paszke et al. *PyTorch: An Imperative Style, High-Performance Deep Learning Library*. Computer Program. 2019.

[9] François Chollet et al. *Keras*. `https://keras.io`. 2015.

[10] Liqun Luo. *Principles of Neurobiology*. Garland Science, Taylor & Francis Group, 2016.

[11] Trevor Huff, Navid Mahabadi, and Prasanna Tadi. *Neuroanatomy, Visual Cortex*. StatPearls Publishing Copyright © 2021, StatPearls Publishing LLC., 2021.

[12] Stephen W. Kuffler. “Disharge Patterns and Functional Organization of Mammalian Retina”. In: *Journal of Neurophysiology* 16.1 (1953), pp. 37–68.

[13] David H. Hubel and Torsten N. Wiesel. “Receptive Fields of Single Neurones in the Cat’s Striate Cortex”. In: *The Journal of Physiology* 148.3 (1959), pp. 574–591.

[14] David H. Hubel and Torsten N. Wiesel. “Receptive Fields, Binocular Interaction and Functional Architecture in the Cat’s Visual Cortex”. In: *The Journal of Physiology* 160.1 (1962), pp. 106–154.

[15] Fred Attneave. “Some Informational Aspects of Visual Perception”. In: *Psychological Review* 61.3 (1954), pp. 183–193.

[16] Horace B. Barlow. *Possible Principles Underlying the Transformation of Sensory Messages*. Vol. 1. Sensory communication. MIT Press, 1961.

[17] Horace B. Barlow. “Unsupervised Learning”. In: *Neural Computation* 1.3 (1989), pp. 295–311.

[18] Eero P. Simoncelli and Bruno A. Olshausen. “Natural Image Statistics and Neural Representation”. In: *Annual Review of Neuroscience* 24.1 (2001), pp. 1193–1216.

[19] David J. Field. “What Is the Goal of Sensory Coding?” In: *Neural Computation* 6.4 (1994), pp. 559–601.

[20] Julia J. Harris, Renaud Jolivet, and David Attwell. “Synaptic Energy Use and Supply”. In: *Neuron* 75.5 (2012), pp. 762–777.

[21] William E. Vinje and Jack L. Gallant. “Sparse Coding and Decorrelation in Primary Visual Cortex During Natural Vision”. In: *Science* 287.5456 (2000), pp. 1273–1276.

[22] Emmanouil Froudarakis et al. “Population Code in Mouse V1 Facilitates Readout of Natural Scenes Through Increased Sparseness”. In: *Nature Neuroscience* 17.6 (2014), pp. 851–857.

[23] Michael Weliky et al. “Coding of Natural Scenes in Primary Visual Cortex”. In: *Neuron* 37.4 (2003), pp. 703–718.

[24] Takashi Yoshida and Kenichi Ohki. “Natural images are reliably represented by sparse and variable populations of neurons in visual cortex”. In: *Nature Communications* 11.1 (2020).

[25] Michael Beyeler et al. “Neural Correlates of Sparse Coding and Dimensionality Reduction”. In: *PLOS Computational Biology* 15.6 (2019), e1006908.

[26] Steven L. Brunton and Nathan J. Kutz. *Data Driven Science & Engineering*. University of Washington, 2017.

[27] Zheng Zhang et al. “A Survey of Sparse Representation: Algorithms and Applications”. In: *IEEE Access* 3 (2015), pp. 490–530.

[28] Trevor Hastie, Robert Tibshirani, and Martin Wainwright. *Statistical Learning with Sparsity: The Lasso and generalisations*. CRC Press, Taylor & Francis Group, 2015.

[29] David Knowles. *Lagrangian Duality for Dummies*. 2010. Available at: `https://www-cs.stanford.edu/people/davidknowles/lagrangian_duality.pdf`.

[30] Niall Hurley and Scott Rickard. “Comparing Measures of Sparsity”. In: *IEEE Transactions on Information Theory* 55.10 (2009), pp. 4723–4741.

[31] Emmanuel J. Candes and Terence Tao. “Decoding by Linear Programming”. In: *IEEE Transactions on Information Theory* 51.12 (2005), pp. 4203–4215.

[32] Moshe Leshno et al. “Multilayer Feedforward Networks with a Nonpolynomial Activation Function Can Approximate Any Function”. In: *Neural Networks* 6.6 (1993), pp. 861–867.

[33] Unknown. *Drawing Neural Networks in TikZ: Short Guide*. 2021. Available at: `https://latexdraw.com/drawing-neural-networks-in-tikz-short-guide/`.

[34] Sebastian Ruder. “An overview of gradient descent optimization algorithms”. In: *arXiv preprint* arXiv:1609.04747v2 (2017).

[35] Diederik P. Kingma and Jimmy L. Ba. “Adam: A Method for Stochastic Optimization”. In: *ArXiv Preprint* arXiv:1412.6980v9 (2017).

[36] Benoit Liquet, Sarat Moka, and Yoni Nazarathy. *The Mathematical Engineering of Deep Learning*. 2021. Available at: `https://deeplearningmath.org/index.html`.

[37] Michael A. Nielson. *Neural Networks and Deep Learning*. 2015. Available at: `http://neuralnetworksanddeeplearning.com/faq.html`.

[38] Gavin Weiguang Ding. *Draw Convnet*. 2018. Available at: `https://github.com/gwding/draw_convnet`.

[39] Christos Louizos, Max Welling, and Diederik P. Kingma. “Learning Sparse Neural Networks through L0 Regularization”. In: *arXiv preprint* arXiv:1712.01312v2 (2018).

[40] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. “Deep Sparse Rectifier Neural Networks”. In: *Journal of Machine Learning Research: Workshop and Conference Proceedings* 15 ().

[41] Yann LeCun, Corinna Cortes, and Christopher J.C. Burges. *The MNIST Database of Handwritten Digits*. Dataset.

[42] Alex Krizhevsky. *Learning Multiple Layers of Features from Tiny Images*. Report. 2009.

[43] Lu Lu. “Dying ReLU and Initialization: Theory and Numerical Examples”. In: *Communications in Computational Physics* 28.5 (2020), pp. 1671–1706.

[44] Horace B. Barlow. “Single Units and Sensation: A Neuron Doctrine for Perceptual Psychology?” In: *Perception* 1.4 (1972), pp. 371–394.

[45] Bruno A. Olshausen and David J. Field. “Sparse Coding of Sensory Inputs”. In: *Current Opinion in Neurobiology* 14.4 (2004), pp. 481–487.

[46] József Fiser, Chiayu Chiu, and Michael Weliky. “Small Modulation of Ongoing Cortical Dynamics by Sensory Input During Natural Vision”. In: *Nature* 431.7008 (2004), pp. 573–578.

[47] Alejandro Torrado Pacheco et al. “Rapid and Active Stabilization of Visual Cortical Firing Rates Across Light–Dark Transitions”. In: *Proceedings of the National Academy of Sciences* 116.36 (2019), pp. 18068–18077.

[48] Peter Sadowski. *Notes on Backpropagation*. Available at: `https://www.ics.uci.edu/~pjsadows/notes.pdf`.

[49] Olivia Guest. *Using the Gini Coefficient to Evaluate Deep Neural Network Layer Representations*. 2017. Available at: `http://neuroplausible.com/gini`.