UNIVERSITY
OF OSLO

Master’s thesis

Towards Understanding Robustness of Neural Networks using Local Learning Rules

Håkon Olav Torvik

Computational Science: Physics
60 ECTS study points

Institute of Physics
Faculty of Mathematics and Natural Sciences

Spring 2023

Håkon Olav Torvik

Towards Understanding Robustness of Neural Networks using Local Learning Rules

Supervisors:
Anders Malthe-Sørenssen
Mikkel Lepperød
Konstantin Holzhausen

Abstract

This thesis explores the use of a local learning rule to train the filters of a convolution neural network, to determine if this leads to more robust models than those trained with the famous backpropagation algorithm. While machine learning and neural networks have become important statistical tools, capable of finding complex patterns in large amounts of data, they are also highly vulnerable to adversarial attacks. These attacks are small perturbations to the input data, aimed to fool the model into making incorrect predictions. This is a problem for the use of neural networks in safety-critical applications and is important to understand.

The update rule for the parameters of the network is based on local information, and takes inspiration from biological learning, though it does not aim to be biologically plausible. The neural network I used was a feed-forward network, and related to dense associative memory-models. This relation restricts the architecture I used to models with a single hidden layer. I trained a large number of models, exploring the effects of the various hyperparameters. The best performing models had accuracies of 72%, both for the locally trained models, and the backpropagation-trained models I used to compare them against. I defined a quantitative measure for the robustness of the models, in order to make direct comparisons between models. I found there to be a trade-off where models with high accuracy have low robustness. By training ensembles of models with the same hyperparameters I was able to isolate the effect of each, and find that many of the hyperparameters resulted in this trade-off. This was not seen in the backpropagation-trained models.

While my quantitative definition of robustness fails to fully capture the complex nature of robustness, I can still satisfyingly compare the models and draw concolusions. The behaviour of the robustness of all the models was the same for various ways of measuring the robustness. This was true both when using Fast Gradient Sign Method and the stronger attack, Projected Gradient Descent.

Machine learning models are often described to be black boxes, where the patterns they learn are not understandable from the outside. I found that the locally trained models had much smoother convolution filters that could easily be interpreted as colours or edges, something not seen in the backpropagation-trained models. I also showed while not all of the filters were smooth, the models with the highest robustness had a smaller fraction of these unconverged filters.

Acknowledgements

I would like to show my deepest graditude towards my supervisors, Anders, Mikkel and Konstantin. You have been of invaluable help throughout my work. Especially Konstantin for continuous feedback and guidance along the way.

Next, I would like to extend all of my gratefulness to CS. It has been an amazing two years working from here. In particular, the blue table must annhiliate much of my thankfulness. Many good discusssions have been had around this magnificent piece of wood and bricks.

Sreval peolpe need am extra tanks for making shure my thesis did not end up like this. Carl, William, Hedvig, Andreas, Johan Emil, Pappa, and of course, my supervisors.

An extra thanks goes out to Fredrik and Tor Quiztian, for making sure no progress was made on Thursdays, and to the nerds at Theological Physics, for making sure no further progress was made on Fridays.

Finally, I would like to thank the universe for not collapsing before I managed to finish this thesis.

Contents

Chapter 1

Introduction

Artificial neural networks (ANNs) are used across various fields for pattern recognition and data classification [1]. They can learn nonlinear, complex patterns and achieve high accuracies unmet by other approaches. However, it has long been apparent that ANNs are susceptible to *attacks*, where small and often unnoticeable perturbations of the input data can mislead the model [2, 3]. This vulnerability has raised concerns about the reliability and security of ANNs in real-world applications, for instance, in cancer diagnosing [4] or autonomous vehicles [5]. In both these examples, faulty input from intentionally crafted perturbations or even random noise can lead to misclassifications and incorrect decisions and, in the worst case, have fatal consequences for users of the systems [6].

Another problem with ANNs is that they are *black-box* models, where data goes in and predictions come out, with no way of understanding the decision-making process or how the model arrived at the prediction. This lack of transparency can be problematic in applications where decisions made by these models have significant consequences for individuals, such as in finance, where models are used to approve or decline loan applications [7]. Being able to ask why an application was declined is important, especially as many such models are trained on already biased data [8].

ANNs are statistical tools for studying data, consisting of a set of nodes connected by weights, which are adjusted during training such that the network can make predictions about the data. Training is a process where the model is given similar data, from which it can *learn* trends and structure. While ANNs come in many variations, the most basic model is a fully-connected, Feed-Forward Neural Network (FFNN). These are alternating sequences of linear and non-linear maps called layers and can be used for regression and classification tasks. While there are many ways of training a network, the most ubiquitous algorithm is back-propagation, where a loss function is minimised by iteratively updating the network weights using gradient descent. This algorithm is computationally efficient, easy to work with and extends to more complex architectures. However, backpropagation-trained models become vulnerable to attacks and are non-interpretable [9].

1.1 Local learning

There are several ways of increasing the robustness of a model against adversarial attacks, for instance, including regularisation terms in the loss function [10] or using adversarial training[2]. However, these methods are often computationally expensive, leading to decreased accuracy on clean data. One approach to mitigate this is using local learning rules instead of backpropagation when training. Local learning (LL) rules are learning mechanisms that allow the nodes to update the weights based on local interactions with neighbouring nodes rather than relying on global error signals, as in the backpropagation algorithm. Such LL rules can be inspired by biological learning in brains, which are natural neural networks. In this thesis, I investigate whether a neural network trained using specific local learning rule is more robust, less susceptible to adversarial attacks, and more interpretable than a neural network trained with backpropagation.

The idea of taking inspiration from biology in the field of machine learning is not new. In fact, the first ANNs were designed to mimic the structure of the brain [11], but modern ANNs have diverged from biological plausibility [12], using global information not accessible in biological systems. The primary goal of machine learning is to develop models that can perform well on a specific task rather than to understand the workings of the brain. However, there has been a renewed interest in biologically-inspired machine learning models in recent years [13]. These models aim to be more robust, interpretable or energy-efficient, taking cues from the structure and function of the brain to develop algorithms that are harder to fool, easier to understand, or expend less energy. It is important to note that these models are not necessarily intended to be biologically accurate but rather be further inspired by biological mechanisms to make better machine learning models.

One such model is dense associative memory (DAM) [14]. It is a generalisation of Hopfield networks and is sometimes called modern Hopfield networks. The original Hopfield network (HN), first published in 1982 by J. J. Hopfield [15], is a recurrent neural network and a type of spin glass. It was inspired by the Ising model, famous in theoretical physics for exhibiting phase transitions from very simple rules [16]. While capable of solving problems in disparate fields [17], HNs are typically used for pattern recognition or associative memory, where the input is compared to a set of stored *memories*, and the state of the network converges to the memory that is most similar to the input. This can be reconstructing an incomplete pattern or image or recalling a memory from a cue. This associative memory recollection is also observed in biological systems.

In their paper, Krotov and Hopfield also showed the existence of a duality between DAMs and FFNNs with a single hidden layer. In two follow-up papers, they presented an unsupervised training scheme for the FFNNs [18], heavily based on biological learning principles, and showed how to make the networks convolutional [19]. Convolutional neural networks (CNN) are a type of ANNs specifically designed to process data where neighbouring features are more highly correlated than distant ones, such as images or time-series data. They are typically used in image recognition and processing. CNNs trained with the Krotov-Hopfield rule have been shown to be more robust against simple attacks than similar models trained with backpropagation [20]. However, they have not been shown to be more robust against stronger, adversarial attacks.

1.2 Goals, and Structure of the Thesis

It is such CNNs, trained with the Krotov-Hopfield LL rule, that will be the object of study in this thesis. In particular, I will study the robustness of these models, trained to classify images, against first-order adversarial attacks such as *FGSM* and *PGD*. I seek to answer if neural networks trained with the Krotov-Hopfield learning rule give more robust models against first-order adversarial attacks than similar networks trained with the backpropagation algorithm. I will also investigate the interpretability of these models and see if this is related to the robustness of the models. In this process, I will explore the effects of the various hyperparameters on the accuracy and robustness of the models, and the variation within models using the same hyperparameters. This will let me determine some properties of robust networks.

This thesis is structured into four parts. In the first part, I provide the necessary background on neural networks and how to train them using gradient descent and the backpropagation algorithm. In the second part, I outline biological learning and how this is borrowed in bio-inspired artificial learning, before I introduce the specifics of the Krotov-Hopfield training scheme and the principles it builds upon. In the third part, I discuss the dataset I will be using, the details of my implementations, and introduce the adversarial attacks. Finally, in the fourth part, I present the results of the experiments before I discuss the results and conclude the thesis.

1.3 List of Abbreviations

Machine learning models

- **ANN** - artificial neural network
- **CNN** - convolutional neural network
- **DAM** - dense associative memory
- **FNN** - fully-connected neural network
- **FFNN** - feed-forward neural network
- **HN** - Hopfield network

Learning rules

- **BP** - backpropagation
- **GD** - gradient descent
- **LL** - local learning
- **SGD** - stochastic gradient descent
- **VGD** - vanilla gradient descent

Activation functions

- **ELU** - exponential linear unit
- **GeLU** - gaussian error linear unit
- **PReLU** - parameterised rectified linear unit
- **ReLU** - rectified linear unit
- **rExp** - rectified exponential

Adversarial attacks

- **FGSM** - fast gradient sign method
- **PGD** - projected gradient descent

Miscellaneous

- **BCM** - Bienenstock-Cooper-Munro (theory)
- **BWD** - Blue Wall of Death
- **KH** - Krotov-Hopfield

Part I

Machine Learning

Chapter 2

Machine Learning and Neural Networks

Both *machine learning* and *neural networks* are considerably broad terms encompassing various methods and models. In this thesis, I will only consider some of these and will in this chapter define and describe the specific types of machine learning and neural networks I will use. There are many resources for understanding machine learning. For more in-depth explanations than are presented in this paper, see [21] or [22].

2.1 Machine Learning

Machine learning is a catch-all term for all statistical methods where a parameterised function $f(x;\theta)$ *learns* parameters θ from data x. The desired goal is for the function to tell us something about the data by determining the values of the parameters θ. When *trained*, the function f_θ is said to be a model for the data, generating a predicted outcome $f_\theta(x) = \tilde{y}$. This learning can either be supervised, unsupervised or reinforced. For supervised learning, the data is labelled with the desired outcome y, and the training usually involves optimising an objective function, $C(f(x;\theta), y)$. When the training is unsupervised, the data x does not include the labels y, and the model learns by finding patterns in the data itself. This thesis will use both of these training paradigms but not the third basic training paradigm, reinforcement learning.

While there are many parameterised functions that fall under the term *machine learning*, I will only consider neural networks. The main reason neural networks are so powerful is that they are so-called *universal approximators*, meaning they can approximate any function to any desired accuracy [23, 24, 25, 26]. The only requirements stipulated in the universal approximation theorems are that the nonlinearities are not polynomials and that the network is sufficiently *tall*, meaning having enough hidden nodes. Theoretically, *enough* here means infinite, while practically, it is a large number dependent on the complexity of the data. Modern networks use many sequential hidden layers instead of very many nodes in one layer, as this is more computationally efficient. Such networks are said to be deep instead of tall. Nevertheless, these networks still meet the requirements to be universal approximators.

To approximate a given function, one has to find the correct weights between the nodes. However, the universal approximation theorems say nothing about how to find the exact parameters. Therefore, one is free to use any algorithm. Various training algorithms incrementally improve the networks until the desired accuracy is achieved. I will use the Krotov-Hopfield scheme, described in chapter 4, and the backpropagation algorithm, described in chapter 3. [1]

2.2 Feed-Forward Neural Networks

In this section, I will describe the anatomy and workings of FFNNs in detail. Specifically, I will start with fully-connected neural networks (FNNs). First, note the difference between the abbreviations FNN and FFNN. The *feed-forward* part means that the networks are structured layer-wise, and the input is passed from one layer to the next. *Fully-connected* means that all nodes in one layer are connected to all nodes in the next. Figure 2.1 shows an example of a four-layer FNN. The neural network is a structured sequence of layers represented by columns of nodes in the figure. The layers are connected by weights, represented by arrows between the nodes. Not shown in the figure is the bias b, a value added to each node, and the activation function, a non-linear function σ.

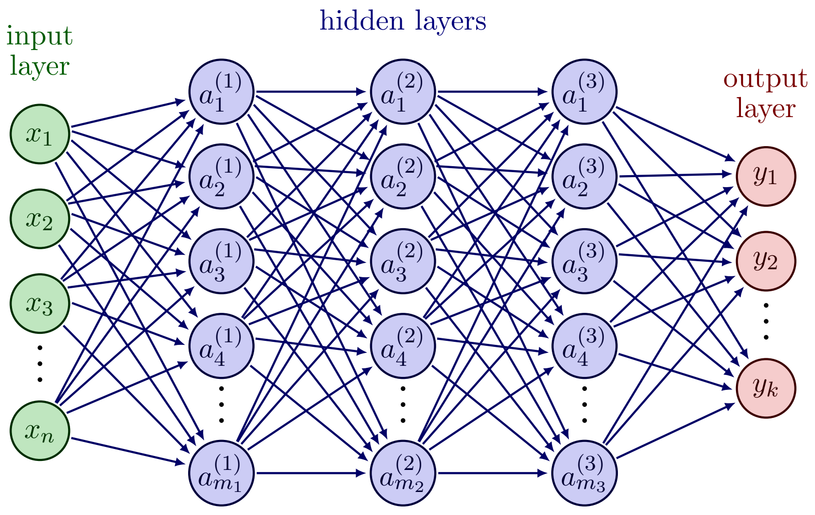
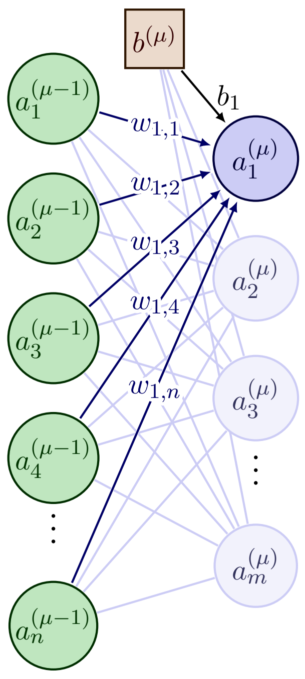


Figure 2.1: A fully-connected feed-forward neural network with three hidden layers. The network takes n-dimensional data and has k-dimensional output. The number of nodes in the hidden layers can be chosen arbitrarily. Circles represent the nodes in each layer. Arrows between the nodes represent the weights. Adapted from [28].

Each layer consists of several nodes, with a node being a single numerical value. The first layer is the input data, usually not counted as a layer in the network itself. Each node in the input layer is a single datum in the dataset one is studying, called features. The following layers are called hidden layers, as the values here are used to compute the value of the next layer but are not directly interpretable as results. Only the values of the last layer, the output layer, are the results of passing data through the network. Neural networks are sometimes called black-box functions, as one can only see what goes in and out of it but not what happens inside. The number of layers and the number of nodes in each layer are so-called *hyperparameters*, set at the discretion of the network architect. The weights are the parameters θ, updated during training.

[1]Interestingly, recent work has shown that there are functions for which no algorithm can exist that lets a neural network approximate it to a desired degree of accuracy [27].



$$
\begin{aligned}
a_1^{(\mu)} &= \sigma\left(w_{1,1}a_1^{(\mu-1)} + w_{1,2}a_2^{(\mu-1)} + \ldots + w_{1,n}a_n^{(\mu-1)} + b_1\right) \\
&= \sigma\left(\sum_{i=1}^{n} w_{1,i}a_i^{(\mu-1)} + b_1\right)
\end{aligned}
$$

$$
\begin{pmatrix} a_1 \\ a_2 \\ \vdots \\ a_m \end{pmatrix}^{(\mu)} = \sigma\left[\begin{pmatrix} w_{1,1} & w_{1,2} & \ldots & w_{1,n} \\ w_{2,1} & w_{2,2} & \ldots & w_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ w_{m,1} & w_{m,2} & \ldots & w_{m,n} \end{pmatrix}^{(\mu)} \times \begin{pmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{pmatrix}^{(\mu-1)} + \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{pmatrix}^{(\mu)}\right]
$$

$$
= \sigma\left[\begin{pmatrix} b_1 & w_{1,1} & w_{1,2} & \ldots & w_{1,n} \\ b_2 & w_{2,1} & w_{2,2} & \ldots & w_{2,n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ b_m & w_{m,1} & w_{m,2} & \ldots & w_{m,n} \end{pmatrix}^{(\mu)} \times \begin{pmatrix} 1 \\ a_1 \\ a_2 \\ \vdots \\ a_n \end{pmatrix}^{(\mu-1)}\right]
$$

$$
a^{(\mu)} = \sigma\left(W^{(\mu)} \times a^{(\mu-1)}\right) = \sigma\left(z^{(\mu)}\right)
$$

Figure 2.2: Illustration of one step in the forward pass. The next layer (blue) takes the value of the activation function (red), evaluated element-wise at the sum of the matrix-vector product of the weights (black) and the previous layer (green), and the bias (brown). The bias is added to each element of the matrix-vector product and can be included in the weight matrix, as shown above. It is also common to not use bias at all. Adapted from [28].

As outlined above, the value of the next layer $a^{(\mu)}$ is a function of the value in the previous layer, $a^{(\mu-1)}$, the weights $W^{(\mu)}$, a bias-term $b^{(\mu)}$, and the activation function σ. The exact relation is derived in figure 2.2, with the full equation being

$$
a^{(\mu)} = \sigma\left(W^{(\mu)}a^{(\mu-1)}\right). \tag{2.1}
$$

In this work, the bias is combined with the weight matrix as the first column, so the bias does not explicitly appear in the equations. This step is shown explicitly in the equations in figure 2.2. The process of computing an output for some input by passing it through all the layers is called the forward pass and takes the form

$$
\begin{aligned}
y = f(x;\theta) &= \sigma_L\left(W^{(L)}\sigma\left(W^{(L-1)}\cdots\sigma\left(W^{(2)}\sigma\left(W^{(1)}x\right)\right)\right)\right), \\
\theta &\equiv \left\{W^{(1)}, W^{(2)}, \ldots, W^{(L)}\right\}.
\end{aligned} \tag{2.2}
$$

Here θ is the set of all parameters of the network. These are what is being updated during training, while the hyperparameters are chosen beforehand. The activation function σ_L used for the output layer usually differs from the one used for the rest, as I will discuss in section 2.4.

A brief note on the mathematical notation in this work: throughout this thesis, there will not be used notation to explicitly differentiate scalar and vector quantities, with both written as lower-case letters. Matrices are written with upper-case letters. Vector elements are lower-case letters with a single index, while matrix elements are upper-case letters with two indices.

Upper-case letters with a single index denote matrix rows. A superscript is an exponent, except when in parentheses, when it is a counter. Some exceptions to these rules might occur when the meaning is otherwise clear from the context.

2.3 Convolutional Neural Networks

When using a fully-connected neural network, one makes very few assumptions about the data it will be trained on. All nodes connect to all the nodes in the previous layer, so all data is visible to every node. The network then has to find any internal structure that can be useful in predicting the output independently. In practice, it has been found to be beneficial to give the network some clue about the structure if there is some. For instance, in images, comparing neighbouring pixels is often more indicative of the object of the image than comparing pixels far apart [29]. This is called an inductive bias and is an essential concept in machine learning [30]. Inductive bias is *a priori* assumptions made by the practitioner.

Convolutional neural networks are a type of neural network that takes advantage of the 2D structure in images. Image processing can often lead to quicker and better results if only a subset of neighbouring pixels are seen simultaneously, such that smaller structures can be identified. This is done with convolutions and is the main feature of convolutional neural networks (CNNs). I will in this section detail how a CNN works and elaborate on some advantages over FNNs when looking at images.

A convolution is a mathematical operation on two functions that express how the shape of one is modified by the other. It is defined mathematically for continuous functions in one dimension as

$$(f * g)(t) = \int_{-\infty}^{\infty} f(\tau)g(t-\tau)d\tau \tag{2.3}$$

This can be interpreted as "sliding" one function over the other and multiplying them at each point. In numerical applications, a discrete convolution has to be used. A small matrix called a kernel is placed on top of the image, and one element of the output is the sum of the element-wise multiplication of the overlapping values. The kernel is then slid over the entire input. In equation (2.3), the input image is f, and the kernel is g. Note that the $-\tau$ in the argument of g means the kernel is flipped when applied to the image. A numerical example of a 2-dimensional discrete convolution is shown in figure 2.3. An image I is convolved with a chosen kernel K. Highlighted in the input image are the values used to calculate the highlighted value in the output image.

In the example convolution in figure 2.3, the kernel, explicitly chosen to extract horizontal edges, is called the Sobel-y kernel [32]. Other kernels can be used to extract different types of information. Using a discrete 2-dimensional Gaussian bell curve as the kernel and applying it to an entire image will blur it. In these cases, a specific kernel is chosen for a particular purpose. Regarding CNNs, the kernel values are the trainable parameters of the network. The network can learn to extract any feature from the data. This is a powerful tool, as it allows the network to learn what features are important for the task. Because each kernel is slid over the preceding layer, they can be much smaller, significantly reducing the number of trainable parameters.

Several hyperparameters decide how the kernel convolves the image. The first is the size of the kernel. For example, the kernel in figure 2.3 is a 3×3 matrix. Larger kernels can extract more complex features but require more parameters to be trained. The second parameter is the stride, the number of pixels the kernel is moved when convolving the image. Neighbouring

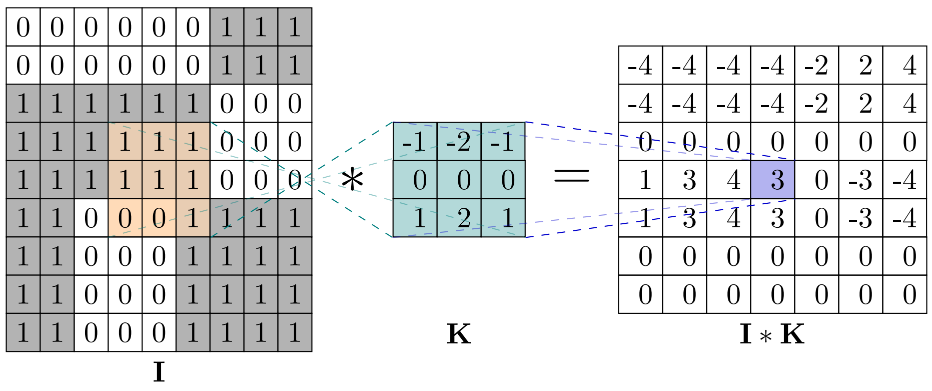


Figure 2.3: Numerical example of 2-dimensional convolution. An input-image I is convolved with the 3×3 Sobel y-kernel K that extracts horizontal edges. Notice that the kernel is flipped when applied to the image. The convolution uses no padding or dilution, and the stride is 1. Adapted from [31].

pixels can be very similar, so increasing the stride is used to decrease the output size without losing important information. A third parameter is padding. Mathematically, a convolution covers all possible positions, even cases where the kernel is outside the input image. Doing this numerically requires having a value for pixels outside the image, called padding. A common choice is zero-padding, where 0 is used in all cases. Another is same-padding, where the value on the edge is copied outwards. Other types can also be used, but this is not explored further here. No padding was used in the example in figure 2.3, so the output is smaller than the input. The parameters are often chosen such that the outcome of the convolution has the same size as the input.

One convolution layer has not just one kernel but a set of kernels, in this context usually referred to as filters. The output from applying one of these filters to an input is called a channel, so the outcome of a convolution layer has as many channels as filters. As a result, the output size can be much greater than the input when using many filters. This can be a problem as it is computationally heavy and leads to many parameters that need training in the next layer. A pooling layer is often applied after a convolution layer to solve this. This layer reduces the output size from a convolution layer by combining the values in a channel. The most common types of pooling are max-pooling and avg-pooling, where the maximum or the average value in a region is chosen. The size and the stride of the region are hyperparameters.

As mentioned above, larger filters can extract more complex features. Often, a CNN has many convolution layers, such that the first layers can extract simple features, such as edges, and the later layers can see more complex structures. As with FNNs, activation functions are used between each convolution layer. After the last convolution layer, one or two fully-connected layers are typically used to classify the image.

2.4 Activation Functions

To approximate *any* function, a neural network must be non-linear. A matrix-vector multiplication is linear, and the addition of the bias is also a linear operation. A sequence of linear operations is also linear, so the non-linearity must be introduced by a non-linear activation function σ between the layers, applied element-wise. Historically, common choices of activation functions have been the sigmoid function or the hyperbolic tangent. However, the rectified linear unit (ReLU) [33], and variations of it, has, in the last decade and a half, been part of what has made deep-learning models feasible and successful. These three are given by

$$\text{Sigmoid: } \sigma(x) = \frac{1}{1+e^{-x}}, \tag{2.4}$$

$$\text{Hyperbolic tangent: } \sigma(x) = \tanh(x) = \frac{e^{2x}-1}{e^{2x}+1}, \tag{2.5}$$

$$\text{ReLU: } \sigma(x) = \max(0, x). \tag{2.6}$$

The sigmoid and hyperbolic tangent functions have bounded output domains, $[0, 1]$ and $[-1, 1]$, respectively. This was long seen as beneficial, as it keeps the values in the network small, avoiding large numbers that can cause numerical problems. However, the activation values saturate for large positive or negative values, meaning the differences are small, and the gradients close to zero. The network is thus less sensitive to changes in this region, with the small gradients making it hard to escape this state. The problem of small gradients, leading to no learning, is called the *vanishing gradient* problem. While ReLU is unbounded, which can lead to overflow, it was found to be more effective in practice because its derivative is 1 for all positive values [33].

A large family activation functions similar to ReLU has since emerged. These include the parameterised ReLU (PReLU) [34], the exponential linear unit (ELU) [35] and the Gaussian error linear unit (GELU) [36]. I also include the rectified exponential function (rExp). These are defined by

$$\text{PReLU: } \sigma(x, \alpha) = \begin{cases} \alpha x & x < 0, \\ x & x \geq 0 \end{cases}, \tag{2.7}$$

$$\text{ELU: } \sigma(x, \alpha) = \begin{cases} \alpha(e^x - 1) & x < 0, \\ x & x \geq 0 \end{cases}, \tag{2.8}$$

$$\text{GELU: } \sigma(x) = x\Phi(x); \quad \Phi(x) = P(\chi \leq x), \chi \sim N(0, 1), \tag{2.9}$$

$$\text{rExp: } \sigma(x) = \max(0, e^x - 1). \tag{2.10}$$

All the activation functions discussed here are shown in the domain $[-4, 4]$ in figure 2.4. ReLU is 0 for all negative input values. The derivative here is also 0, so nodes with negative values will get no training. PReLU fixes this potential issue by having a small slope α for negative values. The value of α can either be a hyperparameter or a trainable parameter. ELU is similar to PReLU but has an exponential slope and approaches $-\alpha$ as $x \to \infty$. For ELU, α is usually taken to be 1. GELU is a smooth implementation of the ReLU function. $\Phi(x)$ in equation (2.9) is the cumulative distribution function of the standard normal distribution. ReLU, PReLU and ELU are all linear for positive input, and GeLU is roughly so. rExp is exponential, which may seem like an odd choice of activation function, as it can easily lead to overflow. I include it in this work because the exponential function's power-series expansion is an infinite sum of polynomials with increasing power. As mentioned in the introduction of this thesis and explored further in chapter 4, the memory capacity of a DAM grows with the power of the energy function. rExp is, therefore, a potentially interesting activation function for the Krotov-Hopfield models.

Usually, all the layers have the same activation function except for the output layer. The correct activation function for the output layer depends on the purpose of the neural network.

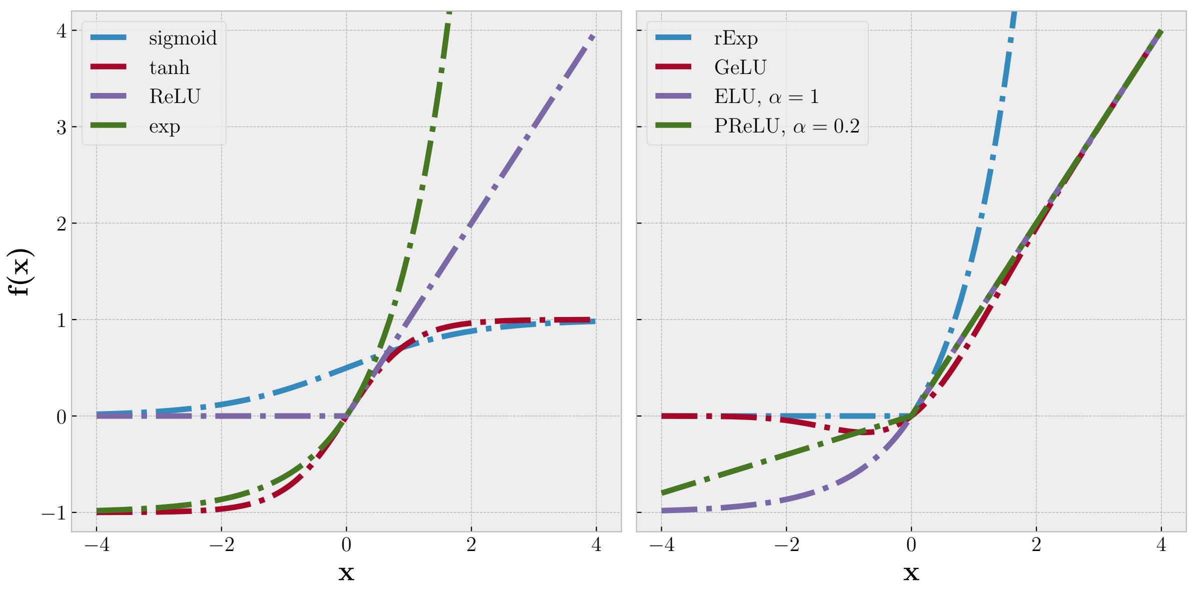


Figure 2.4: The activation functions discussed above in the domain $[-4, 4]$. Their derivatives are shown in figure 3.1

When doing regression, i.e., predicting a function's value at some input point x, one usually does not use an activation function, as the node's value is the prediction. In binary classification cases, i.e. when predicting whether an input belongs to one of two classes A and B, one uses the sigmoid function, as it outputs a value in the interval $[0, 1]$, which can be interpreted as the probability of the input belonging to class A. Finally, when doing multi-class classification, i.e. predicting which of k classes an input belongs to, one wants a value for each class, and for this, the softmax function is used. It is given by

$$\sigma(z)_i = \frac{e^{z_i}}{\sum_{j=1}^{k} e^{z_j}}, \quad i = 1, \dots, k. \tag{2.11}$$

While the other activation functions discussed above work element-wise on a layer, and the activation value of each node is independent of the others, the softmax function does not give independent outputs. Instead, it considers the value in each node and outputs values between 0 and 1 for each class, which can be interpreted as the probability of the input belonging to each class. The node with the highest value is the predicted class. It must be noted that these are not actual probabilities because at no point in the frequentist approach outlined here has uncertainties or the distribution of data been taken into account. The softmax function merely outputs positive numbers that sum to 1.

2.4.1 Weight Initialisation

In addition to vanishing gradients, *exploding gradients* is a common problem in machine learning, where the gradients diverge instead of converging towards 0. These two issues have similar causes [37], and can both be mitigated by careful weight initialisation. The weights are initialised randomly, usually from the standard normal distribution. However, this makes the variance too large, which can cause gradient issues. The favourable distribution and variance depend on the activation function used [38, 34]. Xavier-initialisation is used for symmetric functions like sigmoid and tanh, while He-initialisation is used for ReLU and its variants.

Chapter 3

Supervised Training of Neural Networks

The previous chapter described how the forward pass in fully-connected and convolutional neural networks works. However, with the parameters being initialised randomly, the output of a network would also be random and not useful for anything. Therefore, the network has to be trained. In this section, I will explain gradient descent and the backpropagation algorithm. This is not the algorithm I will use to train the models of primary interest in this work, but the models I will compare them against. I will explain how a local learning rule is used to train models for classification in chapter 4.

3.1 Gradient Descent

I will start with a few definitions needed to understand the problem of finding good parameters for the model f, before detailing gradient descent, which is the idea that lets us solve it. Then finally, I will explain the backpropagation algorithm, which is the method used to update the parameters for gradient descent.

When doing supervised training, one needs a labelled training dataset, $\{X, y\}$, and a parameterised, differentiable model f, like a neural network. $X = \{X_1, X_2, \ldots, X_N\}$ is the input, containing N data-points. Each data-point X_i consists of several features. In the context of image classification, each data point is one image, and each feature is a single pixel value. $y = \{y_1, y_2, \ldots, y_N\}$ is the class label for each data point, which must be set manually.

The central goal of any type of machine learning is finding patterns in the dataset, such that the model f_θ can give correct predictions about new data, X', which the model has not seen before. This is done by finding parameters θ which minimises the loss of X, given by

$$\mathcal{L}_i(X_i) = \mathcal{C}\Big(f(X_i;\theta),\ y_i\Big),$$

where $\mathcal{L}_i$, a scalar, is the loss for a single data point, and $\mathcal{C}$ is a chosen cost function. The total loss of X is the average of all losses,

$$\mathcal{L}(X) = \frac{1}{N}\sum_{i=1}^{N}\mathcal{L}_i = \frac{1}{N}\sum_{i=1}^{N}\mathcal{C}\Big(f(X_i;\theta),\ y_i\Big). \tag{3.1}$$

The hope is that if X' is similar to X, and $\mathcal{L}(X)$ is small, then $\mathcal{L}(X')$ is also small, and $f_\theta(X') = \tilde{y}'$ is close to y', the true labels. Here, *similar* means being drawn from the same distribution and domain.

The cost function serves as a measure of how well the model is performing at the desired task. It is used to quantify the difference between predicted and true output, y' and y. Several functions are commonly used, depending on the type of problem. Mean squared error (MSE) is a common choice for regression problems, while for classification problems, cross-entropy is a popular choice and the one I will use. The cross-entropy between two probability distributions p and q is defined as

$$\mathcal{C}(p, q) = -\sum_{i=1}^{k} p_i \log q_i, \tag{3.2}$$

where k is the number of classes. p is the true label, and q is the predicted label. In the rest of the chapter, I will keep the cost function general, denoted $\mathcal{C}$.

Expressing the ultimate goal in this mathematical framework, achieving small loss requires finding parameters θ that minimises $\mathcal{C}(f_\theta(X), y)$. The cost function $\mathcal{C}$ is minimised when $f_\theta(X) = y$, meaning that all predictions are exactly what is expected in the training dataset. Because neither the data X nor the labels y can be changed, the loss is a function of the parameters θ alone. Therefore, the loss can be written only as a function of θ, which gives the following optimisation problem

$$\min_\theta \mathcal{L}\Big(f(X;\theta), y\Big) \to \min_\theta \mathcal{L}(\theta). \tag{3.3}$$

3.1.1 Optimisers

Gradient descent is an iterative algorithm that solves this by updating the parameters θ in the direction of the negative gradient of the loss function, as this points in the direction in the parameter space that will most quickly decrease the loss. The most straightforward optimiser that solves equation (3.3) is called 'vanilla gradient descent (VGD) and has the update rule

$$\theta_{t+1} = \theta_t - \eta \nabla_\theta \mathcal{C}(\theta_t), \tag{3.4}$$

where η is the length along the gradient vector, the parameters are updated, called the learning rate, and θ_0 is initialised randomly. The updates are performed iteratively until either the loss is sufficiently small or the gradient has converged, meaning being in a flat region of the loss landscape. In practice, it is run for a predetermined number of iterations, N_t. Each iteration is called one epoch.

With N_θ parameters in the model f, the loss landscape is an N_θ-dimensional space. This number can differ by many orders of magnitude depending on the type of model used. Linear regression uses 2, the slope and the intercept. State-of-the-art image classification models and language models can have billions of parameters. The models I will implement in this thesis will have a couple of hundred thousand parameters. The high-dimensional space leads to a highly complex loss landscape. Thus, the GD algorithm is not guaranteed to find the global minimum, and instead, a local minimum, or worse, gets stuck in a saddle region. However, while it is highly unlikely to find the global minimum, it has been found that the loss landscape has many deep valleys with flat bottoms [39]. Thus, there are broad areas that yield reasonably good performance. Regularisation techniques can help smoothen the loss landscape further, increasing the chance of finding good local minima without overfitting the model to the training data.

Modifying the update rule in equation (3.4) gives other optimisers that can improve the final result or speed up convergence. Stochastic gradient descent (SGD) is one such variant, where the true gradient of the loss on the entire dataset X is approximated by using only a sampled subset of the data at a time, X_b, called a minibatch. One epoch consists of B updates of the parameters θ instead of one, where B is the number of minibatches. The sampling is done without replacement, such that

$$X = \bigcup_{b=1}^{B} X_b.$$

While at each step of SGD, the gradient is only an approximation of the true gradient for the entire dataset, the parameter update happens more often. This means that the algorithm can adapt to the loss landscape faster and thus converge more quickly. Also, the gradient is faster to compute for smaller batch sizes, further speeding up computation. The batch size must be sufficiently large, such that the batch is a representative distribution of the entire dataset, so one must consider a trade-off between computational efficiency and a good approximation of the true gradient. Another reason to use minibatches is that many datasets are too large for the computing device to keep in memory at a time. Hence, this method allows for training models on arbitrarily large datasets.

Other popular optimisers change the update rule in equation (3.4) more drastically, often retaining information about previous steps. One such is SGD with momentum, which adds a parameter termed *inertia* to calculate a weighted average of the current and previous gradients to update the parameters. This helps the algorithm avoid getting stuck in flat regions of the loss landscape. The update rule looks like

$$\begin{aligned} v_{t+1} &= \gamma v_t + \eta \nabla_\theta \mathcal{C}(\theta_t), \\ \theta_{t+1} &= \theta_t - v_{t+1}, \end{aligned}$$

where γ is the momentum parameter, and v_t is the 'velocity' of the update. The momentum is the first-order moment of the gradient.

Adaptive Momentum, of Adam [40], for short, is another popular optimiser that uses the first and second-order moments of the gradient to update the parameters. The optimiser algorithm I will use is AdamW, a version of Adam that does the weight decay slightly differently from the standard Adam [41]. Weight decay is a regularisation method that penalises large weights and is often used to prevent overfitting. Overfitting is the phenomenon when the model f_θ learns the training data X too well, so that even though $\mathcal{L}(X)$ is small, and X' is similar to X, $\mathcal{L}(X')$ is large. This is because the model has learned the noise in the training data, not the underlying structure. This is a problem because the model will not generalise well to new data and, therefore, be useless. To prevent this, optimisers include regularisation, which penalises large weights. While the Adam algorithm incorporates the weight decay in the weighted average from previous steps, the AdamW optimiser only does it in the update rule and should lead to better generalisation. This assumption was not tested in this thesis, as it is outside the scope of research here.

The update rule for AdamW is given in algorithm 1. m and v are the first and second moments of the gradient, respectively. Both are initialised as 0. $\hat{m}$ and $\hat{v}$ are bias-corrected versions of m and v, respectively, to account for the fact that they are approximated as weighted averages. β_1 and β_2 are the hyperparameters that set the memory lifetime of the moments. ε is a small number to avoid division by zero. λ is the weight decay parameter. While this algorithm adds many new hyperparameters, it is common to keep them at the recommended values and only tune η. These recommended values are $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\varepsilon = 10^{-8}$, and $\lambda = 0.01$.

Algorithm 1 AdamW update rule, at epoch t

```
$g_{t+1} \leftarrow \nabla_{\theta_t} \mathcal{C}(\theta_t),$    ▷ Finding the gradient
$m_{t+1} \leftarrow \beta_1 m_t + (1 - \beta_1) g_{t+1},$    ▷ First moment
$v_{t+1} \leftarrow \beta_2 v_t + (1 - \beta_2) g_{t+1}^2,$    ▷ Second moment
$\hat{m}_{t+1} \leftarrow m_{t+1}/(1 - \beta_1^t),$    ▷ $\beta_1$ is taken to the power of $t$
$\hat{v}_{t+1} \leftarrow v_{t+1}/(1 - \beta_2^t),$    ▷ $\beta_2$ is taken to the power of $t$
$\theta_{t+1} \leftarrow \theta_t - \eta_{t+1} \left[\hat{m}_{t+1}/\left(\sqrt{\hat{v}_{t+1}} + \varepsilon\right) + \lambda\theta_t\right].$    ▷ The parameter update
```

3.1.2 Learning Rate Scheduling

Notice that in algorithm 1, the learning rate η is given a subscript, as with the other parameters. It is not constant during training but a function of the current step t. This function is called the learning rate scheduler. Learning rate scheduling is a technique used in machine learning to adjust the learning rate during the training process. The learning rate determines the step size taken in the gradient direction during gradient descent. Setting the learning rate too high can result in oscillations or divergence while setting it too low can result in slow convergence or getting stuck in local minima. Learning rate scheduling aims to balance these two extremes by gradually decreasing the learning rate over time as the model converges.

Several approaches to learning rate scheduling include step decay, exponential decay, and cosine annealing. In step decay, the learning rate is reduced by a fixed factor at specific intervals, such as after a certain number of epochs or when the validation error plateaus. In exponential decay, the learning rate is reduced exponentially over time, which can be more effective in achieving a smooth and gradual reduction. Finally, cosine annealing is another approach where the learning rate is decreased following a cosine function, resulting in a cyclical pattern of increasing and decreasing the learning rate over training.

Learning rate scheduling can improve the performance of deep learning models by enabling them to converge faster and achieve better accuracy. However, the optimal learning rate schedule can depend on various factors, such as the model's architecture, the dataset's size, and the task's complexity. Therefore, experimenting with different scheduling techniques and hyperparameters is often necessary to find the most effective approach for a given problem.

3.2 Backpropagation

To recap what I have done in this chapter so far, I have introduced the loss $\mathcal{L}$ and cost-function $\mathcal{C}$, learning rate scheduling, and different optimisers used to update the parameters θ such that the loss decreases, using the gradient of the loss. But the gradient must be calculated w.r.t all the parameters θ_i at every step t. The algorithm used to do this is called the backpropagation algorithm. It is a recursive algorithm that uses the chain rule to calculate the gradient, starting at the back of the network and working its way forward. I will in this section describe how it works.

First, by recalling equation (2.2) and figure 2.2, the activation-value for layer μ of a FNN is

$$a^{(\mu)} = \sigma(z^{\mu}) = \sigma\left(\sum_{i=0}^{n} W_i^{(\mu)} a^{(\mu-1)}\right), \tag{3.5}$$

where n is the number of nodes in layer μ, and σ is the activation function. $W_{0,k}$ is the bias, and $a_0 = 1$. Going further, I will not treat the bias as a special case. In general, W_{ij}^{l} is the weight connecting the i-th node in layer $l-1$ to the j-th node in layer l. $l = L$ is the output layer.

The equations for the gradient of the loss w.r.t the weights in layer μ are given by

$$\frac{\partial \mathcal{L}}{\partial W^{(\mu)}} = \frac{\partial \mathcal{L}}{\partial z^{(\mu)}} \cdot \left(a^{(\mu-1)}\right)^T, \tag{3.6}$$

where the gradient of the loss w.r.t the pre-activation values in layer μ is

$$\frac{\partial \mathcal{L}}{\partial z^{(\mu)}} = \left(W^{(\mu+1)}\right)^T \cdot \frac{\partial \mathcal{L}}{\partial z^{(\mu+1)}} \cdot \sigma'(z^{(\mu)}). \tag{3.7}$$

These two equations are true for all layers except the last one, $\mu = L$, and can be used to propagate the error forwards, starting at the back of the network. For the last layer, the gradient of the loss w.r.t the pre-activation values is

$$\frac{\partial \mathcal{L}}{\partial z^L} = \frac{\partial \mathcal{C}}{\partial a^L} \cdot \frac{\partial a^L}{\partial z^L}, \tag{3.8}$$

where a^L is the output of the softmax activation function, and the second factor is the derivative of it. $\sigma'(z)$ in equation (3.7) is the derivative of the activation function between the hidden layers, as described in section 2.4. The second factor in equation (3.8) is the gradient of the activation function of the output layer, usually the softmax function.

By collecting all the terms above, the full gradient of the loss can be written as

$$\nabla_\theta \mathcal{C}(\theta) = \left[\frac{\partial \mathcal{L}}{\partial \theta_1}, \frac{\partial \mathcal{L}}{\partial \theta_2}, \cdots \frac{\partial \mathcal{L}}{\partial \theta_N}\right]^T = \left[\frac{\partial \mathcal{L}}{\partial W^{(1)}} \parallel \frac{\partial \mathcal{L}}{\partial W^{(2)}} \parallel \cdots \parallel \frac{\partial \mathcal{L}}{\partial W^{(L)}}\right]^T \tag{3.9}$$

where equation (3.6) is used to calculate each term. This is the output of the famous backpropagation algorithm, a simple, recursive algorithm that has come to control a significant part of human lives through the magic of the chain rule. It is the backbone of supervised training, which is summarised in algorithm 2.

While algorithm 2 is very general and is valid for supervised training of all types of machine learning models, equations (3.6) to (3.9) are specific to FNNs. In CNNs, the equations must be altered somewhat to reflect the convolution. If using parameterised activation functions, the gradient must be calculated w.r.t these. Other architectures doing other operations will require different equations. To calculate the gradient for all these types of operations, frameworks implementing *automatic numerical differentiation* (autograd) keep track of all operations performed on the quantities, such that the gradients can be calculated by re-tracing them. This also works for operations that are not differentiable, such as the max-pooling operation in CNNs. I will not detail how autograd works in detail in this thesis. For this, see, for instance, [42].

Figure 3.1 shows the derivatives of the activation functions shown in figure 2.4. Observe that the maximum value of the gradient of the sigmoid function is small compared to the others.

Algorithm 2 Supervised training of an FFNN

```
given data $\mathcal{D} = \{(\mathbf{x}_i, y_i)\}_{i=1}^{N}$

initialise network $f(\theta_0; x)$, with $\theta_0 \sim \mathcal{N}(\sigma, \mu) \in \mathbb{R}^N$

chose learning rate $\eta$, epochs $T$, cost function $\mathcal{C}$

for $t = 0$ to $T$ do
    $\mathcal{L} \leftarrow \sum_{i=1}^{N} \mathcal{C}(f(\theta_t; \mathbf{x}_i), y_i)$

    $\nabla_{\theta_t} \mathcal{C}(\theta_t) \leftarrow \text{backpropagate}(\mathcal{L}, \theta_t)$

    $\theta_{t+1} \leftarrow \text{optimiser}(\nabla_{\theta_t} \mathcal{C}(\theta_t), \eta(t))$

end for
```

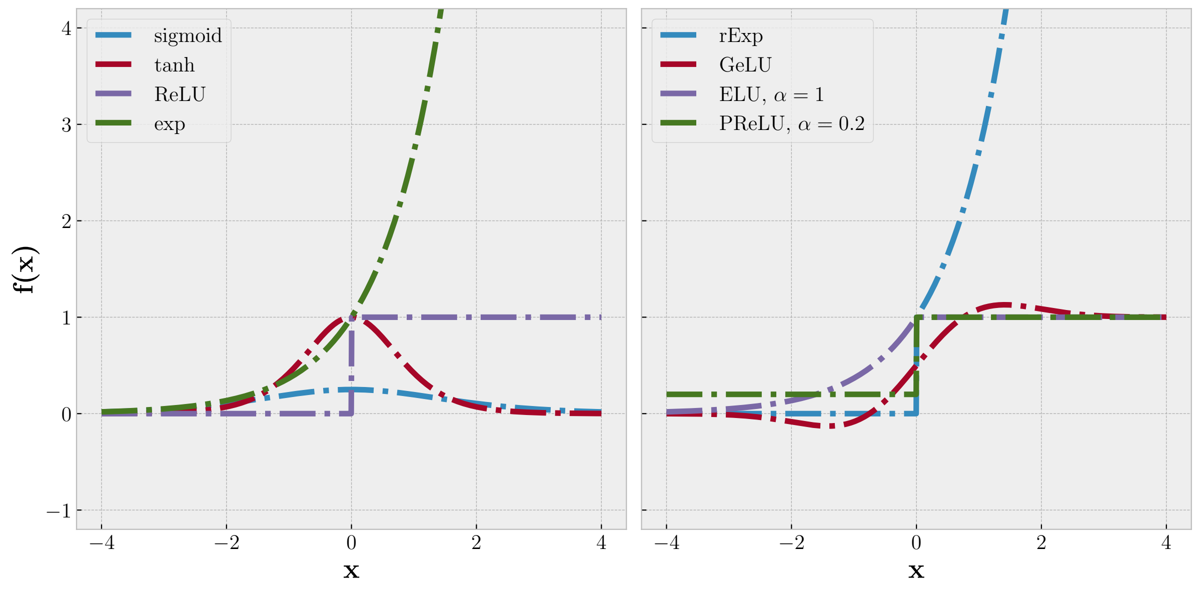


Figure 3.1: The derivatives of the activation functions discussed in section 2.4 and displayed in figure 2.4.

This is one of the reasons why the sigmoid function is not used in modern deep neural networks. The gradients are too small, and by the time the backpropagation reaches the first layer, they are near zero, and little learning happens. This problem is known as vanishing gradients. By the same logic, the exponential function and rExp would not be good either, giving too large gradients and easily causing overflow. This problem is known as exploding gradients. However, the models I use in this thesis are not very deep and should be safe from these problems.

Part II

Local Learning

Chapter 4

Local Learning rules

In the previous chapter, I detailed how neural networks are trained using gradient descent and backpropagation. Here the weight updates rely on global error signals, and the update of one parameter is dependent on the value of all the other parameters in the model. The alternative is local learning rules, where the weight update is only based on local information, for instance, the value of neighbouring neurons, or a Hebbian learning approach.

4.1 Hebbian learning

Hebbian learning refers to a hypothesis of biological learning by D. Hebb, which is summarised in the famous adage ”Neurons that fire together, wire together” [43]. It means that the change in the strength of a synapse is dependent on the timing of the pre- and post-synaptic spikes. Synapses are biochemical connections in the brain which transmit signals between two neurons. These signals are voltage spikes, which propagate down the body of the neuron. While it is not fully understood how information is transported in the brain, the rate of these spikes are generally accepted to be an important piece of the puzzle [44].

The neural networks described in chapter 2 are activation based, which means that each node, or neuron, takes a single value. This is a simplification of rate-based models, where the activation value corresponds to the firing rate. Some models incorporate these rates explicitly with spikes, such as the Leaky Integrate and Fire-model [45]. In activation-based models, Hebbian learning refers to rules where the weight update is a function of the activation in the layer before and after the weights. I will in this chapter start by exploring some mathematical models based on the Hebbian learning principle before presenting the Krotov-Hopfield scheme that I will use in my models.

4.1.1 Oja’s rule

Oja’s rule [46] is a computational form of Hebbian learning for one output neuron y. The activation of that neuron, for some input vector x, is written as

$$y(x) = \sum_{j=i}^{m} x_j w_j,$$

where w are the weights. This is the same as the forward pass in ANNs, as described in chapter 2. The training is Hebbian, with the weight update at time step n given by

$$\Delta w = w_{n+1} - w_n = \eta y_n (x_n - y_n w_n),$$

where η is the learning rate. Oja showed that with this update rule, the weight vector w converges to the first principal component of the dataset. Generalising Oja's rule for N output neurons, it becomes the Generalised Hebbian Algorithm [47]. Now, the weight matrix converges to the N principal components of the data. The extraction of principal components makes these rules applicable to Principal Component Analysis (PCA). It is also related to the Krotov-Hopfield scheme, as will be seen in section 4.3.

4.1.2 BCM Theory

Another relevant local learning model to consider is Bienenstock-Cooper-Munro (BCM) theory [48]. BCM is a mathematical model where a synapse learns to resemble input that excites it more, which can be used to make feature detectors. It was developed to explain the development of orientation selectivity in the visual cortex of mice [12]. The model is based on the observation that the firing rate of a neuron is dependent on the average activity of the presynaptic neurons. Different from Oja's rule, it adds a variable threshold to allow both strengthening and weakening of synapses. This is known as Hebbian and anti-Hebbian learning. The variable threshold is dependent on the activity of the synapses going to the output neurons, where higher average activity makes the threshold higher. This introduces competition among the synapses, which means that the neuron will only respond to the most active synapses and thus learns to detect features in the input.

Like the Oja rule, BCM is a learning rule for a single neuron. Within a population of neurons trained with this rule, many will learn to detect the same patterns. The model can then not be used to develop feature detectors of variable selectivity. This is because it does not implement any interneuron competition. While the neurons in the Krotov-Hopfield model will converge to similar feature detectors, a winner-takes-all mechanism will ensure a high degree of pattern selectivity, which lets the neurons be used as the first layer of a convolutional neural network.

4.2 Hopfield Networks

The development of the Krotov-Hopfield scheme started with the Hopfield network (HN) [15]. It is a fully-connected, recurrent neural network of N binary-state neurons s_b. An HN with 12 nodes is shown in figure 4.1. At every time step, the neurons take the value of either -1 or 1, depending on whether the weighted sum of inputs is below or above a fixed threshold θ. HNs are used to store memories and, when trained, can recall these again from a cue. This memory storing is done by initialising the state of the network to the memory to be stored before updating the weights until the energy function associated with the state is at a local minimum. This weight update can be done in several ways, one of which is a Hebbian learning rule. The energy function is

$$E = -\frac{1}{2}\sum_{i,j}^{N} \sigma_i T_{ij} \sigma_j, \qquad T_{ij} = \sum_{\mu=1}^{K} \xi_i^{\mu} \xi_j^{\mu}, \tag{4.1}$$

where σ_i is one configuration of all the neurons, ξ is the matrix of stored memories, and K is the number of stored memories. The matrix T ensures the stored memories are distinct. Before any memories are stored, T is the identity matrix.

The problem with HNs is that the storage capacity K, the number of memories that can be stored in the network, is severely constrained. It scales asymptotically with the number of neurons N as $K \propto \frac{N}{\log N}$ [49]. Memories are minima of the energy function. This function is

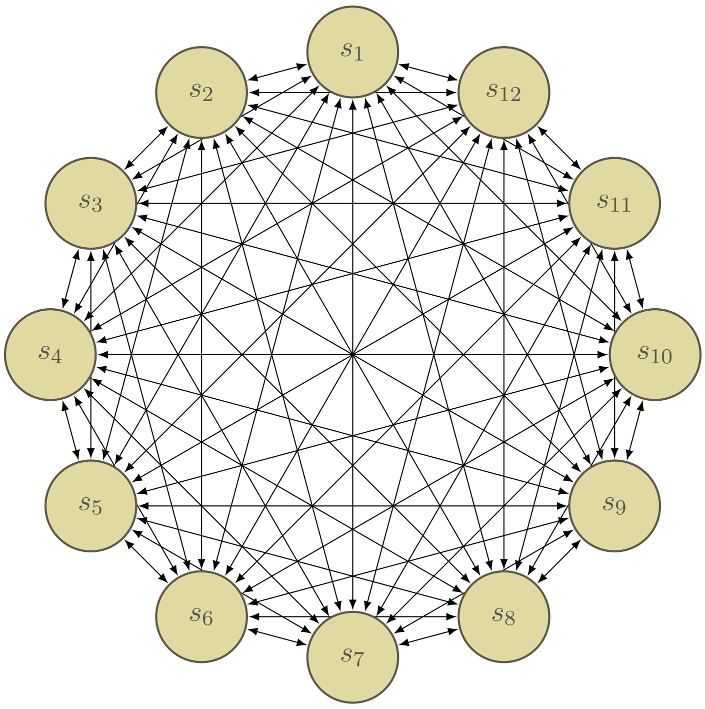


Figure 4.1: A Hopfield network with 12 nodes. The nodes are fully connected, and the weights are symmetric.

quadratic in the neuron values. In 2016, D. Krotov, in collaboration with Hopfield himself, found that the solution to the capacity problem was modifying the energy function to be of a higher order, n, rather than quadratic. This generalisation of HNs leads to the DAMs [14]. The memory capacity of DAMs scales as $K \propto \frac{N^{n-1}}{\log N}$.

In their paper, Krotov and Hopfield also showed the existence of a duality between DAMs and FFNNs with a single hidden layer, where the activation function for the hidden layer is the derivative of the energy function of the DAM. In two follow-up papers, they presented an unsupervised training scheme for the FFNNs [18], heavily based on biological learning principles, and showed how to make the networks convolutional [19].

For these FFNNs to be equivalent to DAMs, their training must be analogous to the memory storing of DAMs. This is based on local learning rules, so backpropagation can not be used. I will in the next section present the unsupervised training scheme invented by Krotov and Hopfield for training these networks.

4.3 Krotov-Hopfield Model

In this section, I will present the local learning rule that is used to train the filters of the convolution layer in the models I will measure the robustness of. These models will henceforth be called LL-models, while the similar models trained with the backpropagation algorithm that I will compare them against are BP-models. Further, the convolution weights of the LL-models will be referred to as synapses to reflect the bio-inspired training and differentiate them from the weights trained non-locally.

4.3.1 Local Learning Principles

The training scheme for the LL-models presented in [18] are based on four local learning principles, which I recount verbatim here due to their importance to the training scheme presented later.

(i) The change of synapse strength during the learning process is proportional to the activity of the presynaptic cell and the function of the activity of the postsynaptic cell. Both Hebbian-like and anti-Hebbian-like plasticity are important. The synapse update is locally determined.

(ii) Lateral inhibition between neurons within a layer, which makes the network not strictly feed-forward during training, is responsible for developing a diversity of pattern selectivity across many cells within a layer.

(iii) The effect of limitations on the strength of a synapse and homeostatic processes will bound possible synaptic connection patterns. The dynamics of our weight-change algorithm express such a bound so that eventually, the input weight vector to any given neuron converges to lie on the surface of a (unit) sphere.

(iv) The normalisation condition could emphasise large weights more than small ones. To achieve this flexibility, we construct (local) dynamics of the synapse development during learning so that the fixed points of these dynamics can be chosen to lie on a Lebesgue sphere with p-norm for $p \geq 2$.

These principles incorporate several ideas from the previous chapters. The first is similar to Hebbian learning, though the second principle expands the definition of "local" to encompass the entire post-synaptic layer, not just pre- / post-synapse. The second principle is similar to BCM theory in that the learning threshold is variable. The competition in this model is between the neurons, not the individual synapses. This leads to diverse feature detectors, which will work well as the first layer in FFNNs. This inhibition is not a feature of biological systems, however. The Krotov-Hopfield model makes no claim to be bio-plausible, unlike BCM theory. Further, while biological neurons are either excitatory or inhibitory, this restriction is not respected in this model and is another source of biological implausibility.

The last two principles concern the neurons' norm when converging. The first states that the synapse vector should converge to unit-norm, while the second establishes a consideration about which p-norm to use, as lower p allows for more equal weights within a single filter. This acts as a regulariser. The value of p is a hyperparameter of the model.

4.3.2 Mathematical Formulation

The plasticity rule that determines the learning dynamics, as presented in [18], is written as

$$\eta \frac{dW_{ki}}{dt} = g(Q) \left[R^p v_i - \langle W_k, v \rangle_k W_{ki} \right], \tag{4.2}$$

where η is the time constant of the learning process, W is the synapse matrix, and each row W_k is the synapse strengths going into one hidden node, or unit, h_k. $g(Q)$ is a non-linear function of the postsynaptic activity. v is the input vector, and R is the radius of the hypersphere the vectors W_k converge to lie on the surface of. The value of R is a free parameter, and $R = 1$ will be used. The inner-product between vectors x and y with elements x_i and y_i is defined as

$$\langle x, y \rangle_k = \sum_{i,j} \eta_{ij}^{(k)} x_i y_j, \qquad \text{with} \qquad \eta_{ij}^{(k)} = |W_{ki}|^{p-2} \, \delta_{ij}. \tag{4.3}$$

Here, δ_{ij} is the Kronecker delta, and p is the chosen Lebesgue norm. For $p = 2$, the inner product is the standard dot-product between two vectors, and for $p > 2$, the metric η_{ij} becomes more critical. The inner product $\langle W_k, v \rangle_k$ is the activity of the postsynaptic neuron h_k.

The rectifying function $g(Q)$ can take both positive and negative values, thus allowing both Hebbian and anti-Hebbian plasticity. It is also used to incorporate lateral inhibition, such that the learning rule satisfy both principle *i* and *ii* of the local learning principles. Considering first *i*, the activity Q in the postsynaptic neuron h_k is defined as

$$Q_k = \frac{\langle W_k, v \rangle_k}{\langle W_k, W_k \rangle_k^{\frac{p-1}{p}}}. \tag{4.4}$$

Then, if the activity is above a threshold Q_*, the function $g(Q)$ should be positive (Hebbian-like), while if below but still positive, it should be negative (anti-Hebbian-like). Otherwise, it should be zero. In the case of $g(Q) = Q$, and with $p = 2$, the dynamics of equation (4.2) would reduce to the Oja-rule from section 4.1.1. Incorporating a temporal competition between the synapses, Krotov and Hopfield define g to push highly driven synapses towards the pattern that activates them, while pushing away slightly driven synapses. This is similar to BCM theory but with competition between neurons, not individual synapses. Keeping the learning rule bounded, the function $g(Q)$ is defined as

$$g(Q) = \begin{cases} 1 & \text{if } Q \geq Q_*, \\ -\Delta & \text{if } Q_* > Q \geq 0, \\ 0 & \text{if } Q < 0, \end{cases} \tag{4.5}$$

where Δ is a small, non-negative number and a hyperparameter of the learning dynamics. Lateral inhibition, principle *ii*, is achieved by letting Q_* be dependent on the activity in all the postsynaptic neurons in the layer. First, Q is calculated for all h_k and ranked, such that $Q_K \geq Q_{K-1} \geq \cdots \geq Q_1$, where K is the number of hidden units. Then, to decrease the computational complexity of the model, g is redefined as a function of ranking index i rather than the activity, as

$$g(i) = \begin{cases} 1 & \text{if } i = K, \\ -\Delta & \text{if } i = K - m + 1, \qquad m \in [2, K] \\ 0 & \text{otherwise.} \end{cases} \tag{4.6}$$

Here, m is a hyperparameter of the learning dynamics, determining which ranked neuron is inhibited. Choosing $m = 2$ will inhibit the second most active neuron, giving more diverse neurons, while higher m will allow some to be more similar, which potentially is beneficial for generalisation and robustness. This redefining of the function g makes the dynamics much faster to compute but is less biologically plausible. It uses the activity of the neurons as the final ranking and allows for the unsupervised training to be done with batches and not just a single image at a time.

It is the second term in equation (4.2) that incorporates principle *iii* and *iv* of the local learning principles by ensuring the convergence of $||W_k||_p$ to R. The synapses are initialised from a standard normal distribution, and the vector has an arbitrary length. It can then be shown that while training, the length converges to R if the two following requirements are satisfied:

$$g(Q) \langle W_k, v \rangle_k \geq 0$$

and

$$||v||_p = R.$$

The way g is defined in equation (4.6), the first constraint is clearly not satisfied. However, Krotov and Hopfield claim that because the violation is weak and limited to the small parameter Δ, the vectors converge as desired. With $R = 1$, the second condition is easily satisfied by normalising the input vectors to unit length in the p-norm before passing it to the model. Thus, each filter in a trained model should have unit length.

Now, re-formulating the learning rule in similar terms as used in chapter 3, equation (4.2) is written as

$$W_{ki}^{(t+1)} = W_{ki}^{(t)} + \eta \cdot g\Big(\text{Rank}(Q_k)\Big) \left[v_i - \left\langle W_k^{(t)}, v \right\rangle_k W_{ki}^{(t)} \right], \tag{4.7}$$

where t is the current epoch. The training that is done here is unsupervised, only dependent on the input. Thus, no optimiser is used. Further, this training method is only possible for a single layer at a time, and the final classifier layers must be trained supervised using back-propagation. This leads to a two-step LL-model training scheme, where the convolution neurons are trained first, and then the classifier layer is trained on top of the convolutional layer. This is done by using the output of the convolutional layer as input to the classifier layer and training the classifier layer to predict the correct class. It has been heuristically found that it is usually much better to train the filters together with the classifier layers so that they can work well together. However, it turns out that this way of training the filters gives good representations, and the classifier layer learns to use these well.

Part III

Implementation & Methods

Chapter 5

Datasets

An important aspect of machine learning is the data. The data is the pre-eminent aspect, as it is the data we want to understand. The models are just techniques to achieve this. However, it can be useful to understand what and how the models are learning independently of the data, as this knowledge gives valuable insight for designing models for new data. This is where *standard datasets* come in, sometimes called *toy datasets*. These datasets have been very well studied and are not very interesting, but they are used to benchmark different models and techniques. As this work aims to investigate the robustness of a trained model, I will be using such datasets as it is easy to compare with other works.

One very common goal of machine learning models is classifying images. Many datasets available for this purpose have been studied in such detail that they are considered standard for benchmarking. One such is the Modified National Institute of Standards and Technology (MNIST)-database [50]. It consists of 60.000 greyscale images of hand-drawn digits, each image being only 28×28 pixels. This is a small and easy dataset, where each image comprises little data, and relatively simple models can be used to classify the numbers correctly to a high degree of accuracy. I mention MNIST not because I will use that to evaluate the robustness but because it was used to test the implementation of the model due to its small size and simplicity. Therefore, it will not receive further attention in this work.

The actual dataset I use in this work is the Canadian Institute for Advanced Research (CIFAR-10) dataset [51]. This is a dataset of 60.000 colour images, each being 32×32 pixels in 3 channels and belonging to one of 10 different classes. The classes are aeroplane, automobile, bird, cat, deer, dog, frog, horse, ship and truck. One example from each of these classes is shown in figure 5.1. This is a more complex dataset than MNIST, as the images are larger and the classes are more diverse. However, it is still relatively small, as 720 pixels is not much data for one image.

CIFAR-10 is also considered to be a quite simple dataset for machine learning. Due to the small size of the images, some images are even hard for humans to classify correctly, and a total accuracy above 94% is considered better than human ability. Many modern machine learning techniques let models achieve accuracies above 99% on this dataset. ImageNet is a more challenging dataset that could lead to better analysis, as it consists of more than 14 million images in more than 20.000 classes. The images are also much larger, with an average resolution of 469×387 pixels in 3 channels, meaning each image has 170 times more data than CIFAR-10 images. However, the limited availability of computing power makes it unfeasible to use ImageNet in this work.

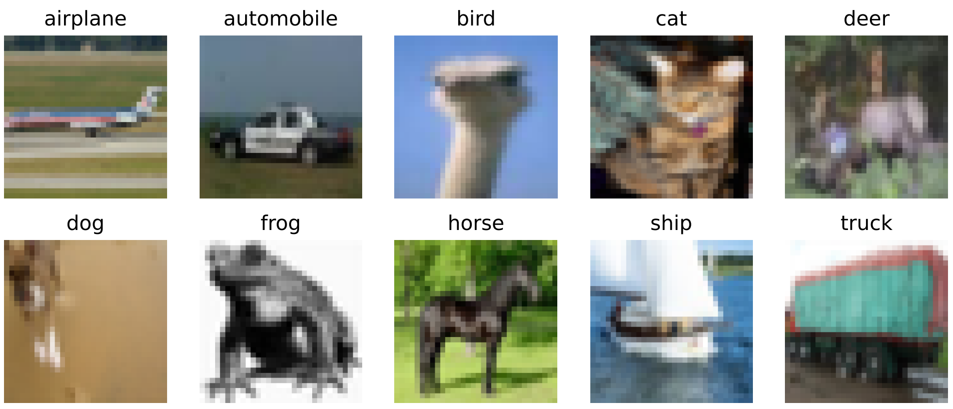


Figure 5.1: Examples of images from the CIFAR-10 dataset.

The CIFAR-10 dataset consists of 50.000 training images and 10.000 test images. The training images are used to train the model, and the test images are used to evaluate the accuracy and robustness of the model, but the models are never trained with these images. The test images are random samples from the total dataset and therefore draw from the same distribution as the training images, as was required in chapter 3 for the model to perform well.

While the examples shown in figure 5.1 were chosen randomly, they display several key features of the images in the dataset. Some images show only the class object, like the cat and truck, while some show some surroundings, like the ship and horse. The dog in this example is off to the side of the image, and the deer is quite hard to identify as a deer. The bird only shows part of a bird, and the frog is in black and white on an empty background. This high variability in the images is pervasive in the dataset and is one of the reasons why it can be hard to train models to get all the images correct.

Each pixel in the images in the original dataset has an RGB value, i.e. three integer numbers between 0 and 255 that represent the intensity of the pixel's red, green and blue colour. When pre-processing the data, I convert these values to floating point numbers between 0 and 1 by dividing all values by 255. This is a typical pre-processing step, as it makes the data easier to work with. One consequence of this is effectively going from discrete data to continuous. A fundamental step in the forward pass of the LL-models is normalising each patch to unit norm, so this is a necessary step. The BP-models do not do this, so they are effectively trained with the discrete data.

Chapter 6

Implementation

I implemented the LL- and BP-models similarly, with the only difference being how the sets of weights are trained. The BP-models are trained end-to-end using backpropagation, while the LL-models are trained in two steps, first just the convolution synapses, then the classifier synapses. This let me train several classifier layers with different hyperparameters for the same convolution filter, significantly reducing training time and storage space. I trained the models for a fixed number of epochs, determined heuristically.

A learning rate schedule was used to decrease the learning rate during training, as described in section 3.1.2. This was done for both the unsupervised and supervised parts of the training for the LL-models. I implemented and tested four different shapes for the learning rate decay. With η_0 being the initial learning rate, t being the current epoch, and T being the total number of epochs, the learning rate was decayed according to the following functions:

$$\begin{aligned} \text{Linear}: \quad & \eta(t) = \eta_0 \cdot \frac{T-t}{T}, && (6.1)\\ \text{Exponential}: \quad & \eta(t) = \eta_0 \cdot e^{-\tau t/T}, && (6.2)\\ \text{Cosine}: \quad & \eta(t) = \frac{\eta_0}{2} \cdot \left(1 + \cos\frac{\pi t}{T}\right), && (6.3)\\ \text{Step decay}: \quad & \eta(t) = \eta_0 \cdot \gamma^{\lfloor t/s \rfloor}, && (6.4) \end{aligned}$$

where τ is the exponential decay rate, γ is a decay factor, and s is the number of epochs between each step decay. These are hyperparameters.

The convolution synapses of the LL-models were not initialised according to the choice of activation function, as described in section 2.4.1. Instead, the initial values are drawn from the normal distribution $\mathcal{N}(\mu, \sigma)$, with μ and σ being hyperparameters. However, to reduce the dimension of the hyperparameter space, these parameters were not tweaked and were kept at a constant $\mu = 0$ and $\sigma = 1$ for all the experiments I performed. The convolution weights of the BP-models, as well as the classifier layers of both models, were initialised according to the He-method, with the initial values drawn from $\mathcal{U}\left(-\sqrt{k}, \sqrt{k}\right)$, where k is the number of nodes in the preceding layer.

The architecture of the two models, which was the same, is shown in figure 6.1. An input image is convolved with the K^2 filters of the model, with each filter being of size $w \times w$. After this, a max-pooling layer is applied, reducing the data amount and emphasising large values. This was done in both [18] and [20]. After the max-pooling layer, the values are passed

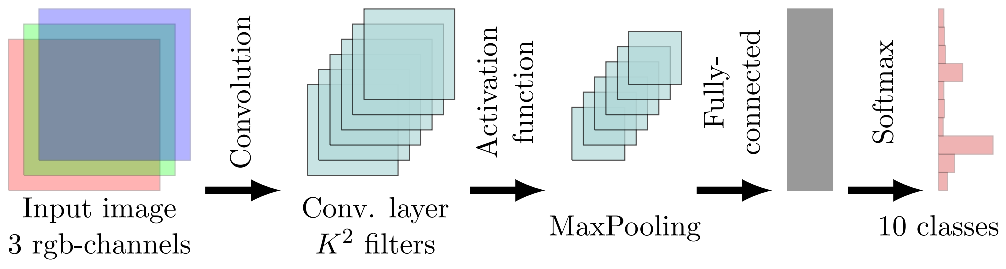


Figure 6.1: Architecture of both the LL-models and BP-models. A convolution layer with K^2 filters is followed by a max-pooling layer, an activation function and a fully-connected classification layer. The output is the "probability" of the input belonging to each of the ten classes, and the prediction is the class with the highest value.

through the activation function. I tried the various activation functions discussed in section 2.4, taken to the power n, where n is the order of the rectified polynomial of the energy function for the corresponding DAM model. Here, the rectified polynomial is $\text{ReLU}(x) = max(x, 0)$, while the other activation functions are smoother variations of it. The activated values are flattened before being passed through a fully-connected classification layer. Then softmax is used to get the "probability" of the input belonging to each of the ten classes. The prediction is the class with the highest value.

As mentioned in the previous chapter, the highest accuracies achieved on the CIFAR-10 dataset are above 99%. The models with this performance are very deep machine learning models, with the number of layers varying from 30 to 200. Due to the origin of the LL-model, as presented in chapter 4, the number of layers is limited to two. Heuristically, deeper is better, so this constrains the expected performance of both the LL and BP-models. However, the focus of this thesis is not to get the best models, but studying how robust the models are against adversarial attacks.

All the code for this work was written in Python, in the machine learning framework PyTorch. This is one of the most popular frameworks used for machine learning. It is very flexible and allows for easy implementation of new models and training methods. The code is publicly available on GitHub[1].

6.1 Patching

The convolution layer in the BP-models is relatively straightforward, as it is a common operation implemented in the PyTorch framework. It involves sliding the filters over the input image and computing the standard inner product between them at each step. The convolution pass in the LL-models is more complicated, as the input vectors must be normalised, and the inner product used is the one defined in equation (4.3). Therefore, I implemented this myself. Specifically, I implemented the extracting of all the sub-images of a single image that are seen by the filters, called patches. While this is a relatively simple process, it is computationally slow, and I tested a few different methods to make it as fast as possible. Because this piece of code would be used for every single forward pass of the model, making it as efficient as possible was desirable.

[1] https://github.com/Haakooto/Masterthesis/tree/main/code

Understanding the shape of the input and output is useful to see how this can be done. The input images are 3-axial tensors, with the first and second axes corresponding to the image's height and width, I_H and I_W. The dataset only has square images, so $I_H = I_W = I$, where $I = 32$. The third axis is the colour channels C, of which there are 3. The output would be 2-axial, with the first axis having $3 \cdot w_H \cdot w_W$ dimensions, with the ws being the height and width of the convolution kernel. I will only work with square kernels, so $w_H = w_W = w$. The second axis contains one dimension for each path. The number of patches N_p from one image depends on the convolution parameters padding and stride, p and s. I only used p= 0 and s= 1. N_p is given by the formula

$$N_p = \left\lfloor \frac{I + 2 \cdot \mathrm{p} - w}{\mathrm{s}} + 1 \right\rfloor^2 = (I - w + 1)^2 \,, \tag{6.5}$$

where I is the height of the image. The shapes are

$$\begin{aligned} \mathrm{size}(x) &= (I, I, C)\,, \\ \mathrm{size}(\mathrm{Patchify}(x)) &= \left(Cw^2, N_p\right). \end{aligned}$$

Practically, this was done for an entire minibatch of images at once, so an extra axis with N_B entries could be included in each of the sizes above.

The first method I tried was flattening the image into one long vector first and then slicing it to access the data. The flattening makes the patches non-contiguous in the vector. The data from one row of the patch is located together, but each row is at a different place. Two for-loops are required to extract the patches, one to find the index of the first row-element and one to go through the rows. Accessing data in a vector like this should be quite fast. However, the arithmetic for determining the start index of each patch and each row in the patch was quite complex, so the first change I made was reordering how the data was extracted. Instead of flattening first, I kept the image in the 3-axial shape. Then, looping over rows and columns, I extracted the patch as a 3-axial slice before flattening it. This is the method used by the PyTorch function `unfold`, and perhaps the most intuitive. This approach was slower but easier to work with. From this, I saw that I could combine the two techniques by first extracting w rows at a time and then flattening these. Before flattening, I permuted the arranging of the columns and channels such that the patches would be contiguous in the flat vector. This method turned out to be about 25% faster than the first method while keeping the simplicity of the second.

From this, one further improvement in performance was made. Instead of double-looping through the images every time, I first created an indexer-array with the same shape as an input image, filled with unique integers. This array was "patched" by the method described above and stored. Now, the patches from one image can be created simply by indexing the image with the indexer-array. No explicit for-loops are needed, and the indexer-array can be created once and reused for every minibatch. Using a kernel size of $w = 4$, this method was four times faster than accessing the data directly, which is a significant improvement. Using larger kernel sizes yielded smaller improvements, with 2.1 times faster for $w = 8$ and 1.6 for $w = 12$. The method of indexer-array could have been done for the first two methods as well, and I believe it would have yielded a similar final performance due to the nature of indexing. However, with a working implementation of the third method, there was no need to test this.

6.2 Model Hyperparameters

As presented in the previous chapters, the models have several important hyperparameters. While I keep the architecture fixed, I will extensively explore the space of hyperparameters to find good models. While both [18] and [20] presented the values for these that they used in their studies, I conducted my own search in the hyperparameter space to understand the effect of each better. I will in this section describe and interpret all the hyperparameters. The LL-models have a few more than the BP-models, and I will clarify which parameters are applicable to both and which are specific to the LL-models. A summary of all the hyperparameters is found in table 6.1.

The first few hyperparameters are the convolution parameters. The models have K^2 filters, chosen to be a square number so that the filters can easily be visualised in a square grid. More filters can increase the number of patterns the model finds in the image, but due to the winner-takes-all implementation in the LL-models, too many filters can lead to many filters that do not converge. The kernel size w decides how complex features can be extracted. Using $w = 32$ is equivalent to not using convolutions. In order to constrain the hyperparameter space, I only use square kernels and only use stride $s = 1$, and no padding, $p = 0$. Some hyperparameters of the LL-models not shared by the BP-models, include m, Δ and p. m is the ranking parameter in the winner-takes-all mechanism, where the m-th most activated neuron is pushed away from the pattern by the small anti-Hebbian learning value $-\Delta$. p is the Lebesgue-norm.

After the convolution, there is a max-pooling layer. This layer's kernel size and stride are hyperparameters, which I will explore the effects of. After this comes the steep activation function. For both models, I will test the different rectifiers discussed in section 2.4, with different powers n. While the steep rectifier is an important part of the LL-models, stemming from the energy function of DAMs, I will also see its effect on the BP-models.

Some hyperparameters relate to the training and not the models themselves. These are the learning rate scheduling, number of epochs and batch size. The convolution synapses of the LL-models are trained separately. Therefore, each of these applies twice to each LL-model. These are at the bottom of table 6.1

The start of my hyperparameter search was informed by the works [18, 19, 20]. The first is the Krotov and Hopfield paper where they derive the training scheme for the feed-forward network, summarised in equation (4.7). They did not use a convolutional model, so I expected the best hyperparameters I found to differ from theirs. The second paper, by Grinberg et al., introduced convolution to the model. However, the architecture differed from mine, as several convolutional layers were placed in front of a single classifier layer. This lets the model use several convolution sizes, extracting features at multiple scales. While the authors showed that this combination of filter sizes improved the accuracy of the model, time did not allow for including such models when testing the robustness in this work. The third paper, by Patel and Kozma, used the same architecture as in this work and tested the robustness of the models against white-box attacks, adding random noise or occluding parts of the image. They did not test the robustness against adversarial attacks, which is the focus of this work. However, they found that the models were more robust to random noise when using the local learning rule. This work seeks to expand on these results.

Table 6.1: Summary of hyperparameters, with domain and valid range. The entire valid range is not interesting for most parameters, and I will only use a subset of the domain. The dividing line in the middle separates the hyperparameters for the convolution layer (top) and the classifier layer (bottom) of the LL-models. For the BP-models, there is no such distinction.

Description	Symbol	Domain	Valid range	Model
Convolution layer				
Convolution filters	K	$\mathbb{N}$	$K > 30$	Both
Convolution Kernel width	w	$\mathbb{N}$	$2 \leq w \leq 32$	Both
Ranking-parameter	m	$\mathbb{N}$	$2 \leq m \leq K^2$	LL-models
Anti-Hebbian learning	Δ	$\mathbb{R}$	$\Delta > 0$	LL-models
Lebesgue-norm	p	$\mathbb{N}$	$p > 2$	LL-models
Classifier layer				
Max-pooling kernel width	w_{pool}	$\mathbb{N}$	$2 \leq w_{pool} \leq 32$	Both
Max-pooling stride	s_{pool}	$\mathbb{N}$	$1 \leq s_{pool} \leq 31$	Both
Activation function	$\sigma(x)$	function	equations (2.6) to (2.10)	Both
Power	n	$\mathbb{R}$	$n \geq 1$	Both
Training				
Number of epochs	N_{epochs}	$\mathbb{N}$	$N_{epochs} > 0$	Both
Batch size	N_{batch}	$\mathbb{N}$	$N_{batch} > 0$	Both
Initial learning rate	η_0	$\mathbb{R}$	$\eta_0 \geq 0$	Both
Learning scheduler	$\eta(t)$	function	equations (6.1) to (6.4)	Both

Chapter 7

Adversarial Attacks and Robustness

The ultimate purpose of this thesis was to quantify the robustness of LL-models and compare this with BP-models. In short, the robustness of a model is its ability to predict the outcome for perturbed input correctly. These perturbations are called attacks, and are generated by attack methods. The next section will define and discuss the attack methods I used to find the adversarial radius for single data point. Then, in section 7.2, I define how I quantify the robustness of the model on the entire test dataset.

7.1 Adversarial Attacks

Attack images refer to images that the model normally classifies correctly, but have been subject to some perturbation π. These perturbations are small, ensuring that the unperturbed and perturbed images appear visually similar, if not entirely indistinguishable, to human eyes. It is imperative that models, like humans, do not easily succumb to such deception. However, when the perturbations reach a certain magnitude, the attack image may significantly deviate from the original clean image. Consequently, if the model is not fooled by large perturbations, it is robust. Any of these perturbations π move the image somewhere in the space of possible inputs. The *adversarial radius*, denoted ρ, is the smallest distance between the original image x and a misclassified attack $x + \pi$. Whereas earlier in this thesis x denoted the training dataset and x' the test dataset, I will now use x for the test set, and x' for the attack set. Any perturbation used in this thesis is additive, so $x' = x + \pi$.

There are two general ways of generating the attack images; *black-box* attacks and *white-box* attacks. In a black-box attack, the model is inaccessible to the attacker, so the perturbations can not be model-specific. Examples are rotations, random additive noise, shadows, and occlusion, where parts of the image are completely hidden. These are quick to calculate but are generally weaker, as they do not target the specific model. In a white-box attack, the model is accessible to the attacker and used against itself. Gradient methods use the gradient of the loss function w.r.t. the input image to maximise the loss. When describing gradient descent in section 3.1, I formulated the training as a minimisation problem,

$$\min_{\theta} \mathcal{L}(f_\theta(x), y). \tag{7.1}$$

Adversarial attacks are the opposite; a maximisation problem, or gradient ascent, now w.r.t. the input x instead of the parameters θ,

$$\max_{\pi} \mathcal{L}(f_\theta(x + \pi), y). \tag{7.2}$$

The simplest such method is Fast Gradient Sign Method. FGSM, as the name suggests, uses the sign of the gradient of the loss. The perturbation is calculated as a step of size ε along the direction that maximises the loss,

$$x' = x + \varepsilon \, \mathrm{sign}(\nabla_x \mathcal{L}(f_\theta(x), y)). \tag{7.3}$$

ε is called the attack-strength, as larger ε takes the attack image further from the original. The attack image is clipped back into the valid domain, $\mathcal{D} \in [0, 1]$. The approximation of the adversarial radius ρ is the smallest ε where the model misclassifies the input. However, this is not necessarily the true adversarial radius. Iterative-FGSM (IFGSM) gives a better approximation, where the perturbation in equation (7.3) is done several times, as

$$x'_{i+1} = x'_i + \varepsilon \, \mathrm{sign}(\nabla_{x'} \mathcal{L}(f_\theta(x'_i), y)), \quad \text{where} \quad x'_0 = x. \tag{7.4}$$

This is done until the model misclassifies the input. The step length ε is now a hyperparameter that needs to be chosen. The number of steps required to reach misclassification is denoted N_I, and the final attack is $x'_{N_I} = x'$. For IFGSM, there are two measures of the robustness for a single attack; N_I and ρ, where the adversarial radius ρ is found as $\rho = ||x - x'||_\infty$.

FGSM and IFGSM are unbounded, meaning the perturbation can be very large. This breaks with the idea of the attack being visually similar to the original image. One way to fix this is to project the perturbation back into a bounded region S_x around the original image x. This leads to the method of Projected Gradient Descent (PGD). It was proposed by Madry et al. in [2], who showed that it was a universal first-order adversary. It is similar to IFGSM, but with the key difference that the perturbation is projected back into the region S_x at each iteration. The perturbation is calculated as

$$x'_{i+1} = \Pi_{S_x} \left[x'_i + \varepsilon \, \mathrm{sign}(\nabla_{x'} \mathcal{L}(f_\theta(x'_i), y)) \right], \quad \text{where} \quad x'_0 = x, \tag{7.5}$$

where Π is the projection operator.

As with IFGSM, both the number of steps required to reach a misclassification and the distance between x and x' are measures of robustness. The region S_x is the l_∞-ball of size s around x. Other l_p-balls could also have been chosen, but the infinity-norm was chosen as it is perhaps the most conceptually simple. The value of every single pixel can vary up to s independently. Defining more comprehensive notions of perceptual similarities between images are beyond the scope of this work, but could be an interesting area of research for future work.

There are several ingredients that can be included or tweaked to produce many flavours of similar attacks. For instance, using the full gradient instead of only the sign. The attack is then true gradient ascent, and runs until it reaches a local maxima. The initial point x'_0 could be chosen randomly inside the S_x-ball, to escape regions of potentially flat loss-landscapes. Random restarts pick many initial points in the S_x-ball, which leads to a more thorough search of the space for further the maximising loss. I could have kept the number of iterations fixed and varied s for the iterative methods. Further, even stronger attacks can be formulated, such as the Carlini-Wagner attack, which uses the second derivative of the loss function. None of these methods will be used in this thesis, but they are worth mentioning and a possible area of research in future work.

7.2 Robustness

The adversarial attacks give a measure of the adversarial robustness ρ for each image in the test dataset. Then, the total robustness of the model $\mathcal{R}$ is a measure of all ρ. For both FGSM and PGD, ρ is determined on only a subset of the test dataset. The selection of this subset is conducted separately for each model. Only images that the model already classifies correctly are used. This makes it easier to compare the drop in accuracy as the attack strength increases across models. Furthermore, only images in which the model displays confidence in its prediction are included, where confidence is quantified by the predicted class "probability" being greater than or equal to some threshold value $C : 0.1 \leq C \leq 1$. This ensures that images which the model is already uncertain about are excluded.

As will be shown in chapter 9, the accuracy of the model as a function of the attack strength ε starts at 100% and drops off as the attack strength increases. The confidence threshold selection has a notable impact on this curve, as the distribution of adversarial radii exhibits a smaller tail on the left side for larger C. For small C, the accuracy decreases exponentially, while for large C, the shape is closer to a logistic function. I chose C sufficiently high to achieve this shape. The robustness of the model $\mathcal{R}$ I defined to be the inflection point of this curve. This is the point where the slope is the steepest, the model most sensitive to perturbations.

In order to get a good value for the inflection point, I fit the measured accuracy to a parameterised logistic function. The generalised logistic function is defined as

$$f(\varepsilon) = 1 - \frac{1}{\left(1 + Qe^{-B(\varepsilon/D - M)}\right)^{1/|\nu|}}, \tag{7.6}$$

where ε is the attack strength, $A(\varepsilon)$ is the accuracy of the model, Q is the steepness of the drop-off, B the growth factor, D a scaling factor, M the midpoint and $\nu > 0$ affect near which asymptote the maximum growth occurs. It has already been assumed the left asymptote is 1 and the right 0. From these parameters determined by fitting, the inflection point, and hence robustness $\mathcal{R}$, is given by

$$\mathcal{R} \equiv \varepsilon_{\text{crit}} = \frac{D}{B}\left(MB + \log\frac{Q}{\nu}\right), \quad \text{where} \quad \left.\frac{d^2 A(\varepsilon)}{d\varepsilon^2}\right|_{\varepsilon=\varepsilon_{crit}} = 0. \tag{7.7}$$

As an example, figure 7.1 shows the generalised logistic function and its inflection point for some sets of parameters. These are only for illustrative purposes. The actual value for each parameter for each model is found by best fit to the data. The fit is a non-linear least squares method, minimising the sum of squared residuals.

As robustness measures adversarial radii, larger values of $\mathcal{R}$ signify more robust models. However, The value does not signify much, as it is an absolute measure. The robustness value only has value when compared against other models. Further, the value determined by the two attack methods, FGSM and PGD, is also not comparable, as PGD is stronger. However, as a sanity check, a more robust model, measured with FGSM, is expected to be more robust when measured with PGD against another model.

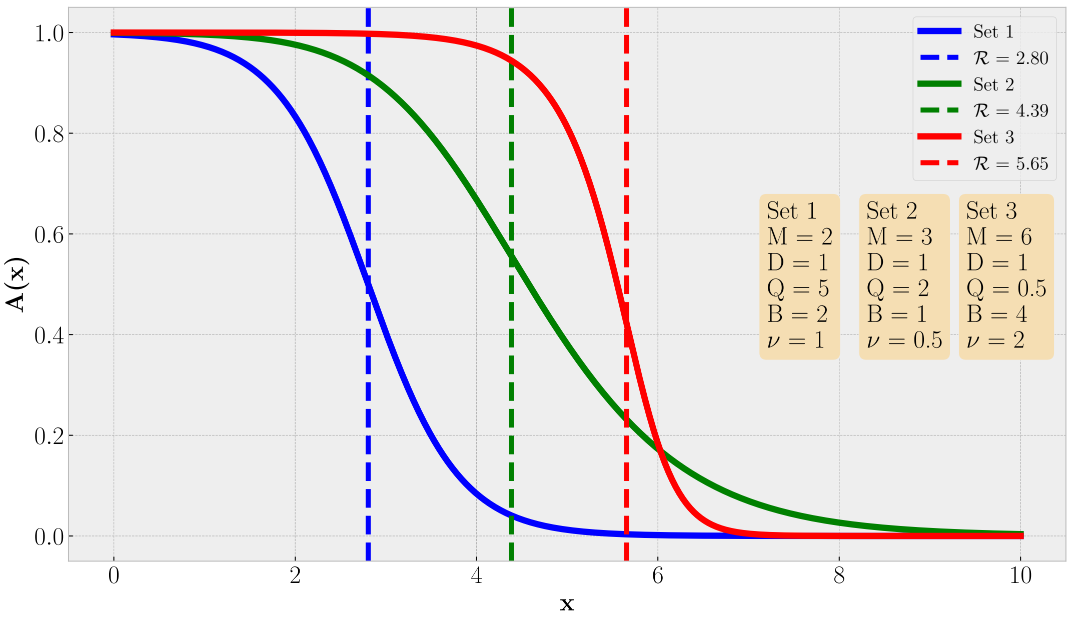


Figure 7.1: Generalised logistic functions with the inflection point marked.

Part IV

Results & Discussion

Chapter 8

Training models

Before applying the attacks to determine the robustness of the models, I trained models with various parameters to find the best-performing models. It is not given that the models achieving the highest accuracy on the dataset will be the most robust. I, therefore, performed various grid searches to have a wide range of models. In this process, I trained over 20.000 models. Some of these were trained in the early stages of my work, and the code was still under development and verification. Therefore, I always trained new models, even though I had previously used a specific set of parameters. These helped me lock the value of some hyperparameters such that I could later span a broader range of the others. In this chapter, I will present my exploration of the model and the search results for the hyperparameter that gives the best models.

8.1 Training LL-models

Because of the two-step training of LL-models, I first focused only on the convolution layer to explore the behaviour of this alone. I trained a model using the same parameters used in [20]. However, they did not report the learning rate used, so for this, I used [19]. Using the parameters $K = 20, m = 2, p = 2, w = 8, \Delta = 0.2, \eta_0 = 1.0 \cdot 10^{-4}$, and a constant learning rate, I trained a model for 1000 epochs, and with batch size 12500. The large batch size was chosen to minimise the training time. A visualisation of the synapses at the end of the training is shown in figure 8.1, which includes labels to indicate specific neurons. This visualisation is created by scaling the vector of synapses going into one neuron such that the largest value is 1, and the smallest is 0, before reshaping it into a $w \times w \times 3$ colour image. This is done for each of the neurons in the model before placing each in a square grid. The resulting visualisation displays what input patch or feature activates each neuron the most. For instance, the neuron shown at location $E1$ in figure 8.1 will be excited by turquoise patches, while $A4$ and $T1$ see vertical stripes and diagonal edges, respectively. The figure is similar to the results shown in the papers [19] and [20].

The neurons in figure 8.1 exhibit various features. Some see pure colours, others see edges or lines in various orientations, while others again are mixed, finding coloured edges, for instance, $H2$. This range of feature detectors displays the winner-takes-all mechanism built into the model, where each neuron specialises in a single feature. This is a desirable property, as it allows the model to learn a wide range of features, not just the most common ones. However, the features are not orientation-invariant. $H5$ and $T12$ are the same, mirrored around the vertical axis. While, on one hand orientation is important for identifying objects, this information could be incorporated into the model differently, reducing the computational load while keeping the capabilities of the model. This was not pursued in this work. Further, there are many quite similar neurons. This could indicate that the number of neurons is too large and that the model could be compressed. This is corroborated by 21 of the neurons that are still noisy, not having converged to a recognisable feature, for instance, $D4$.

To see the process of the random initial state becoming features, figure 8.2 shows the evolution of the neurons at various stages in training. Only the top left quarter from figure 8.1 is included to see the individual neurons more easily. Before the training starts, every neuron looks like random noise. After 40 epochs, two neurons have started becoming features. Ten epochs after this, this number has increased significantly, and at epoch 70, many neurons display clear features. By epoch 150, most are clearly filled in, while the few that have not yet done so by epoch 500. Eight of the 100 neurons displayed remain noisy at epoch 500, and only two fill in by the end of training at epoch 1000. Individual neurons can be studied to see the development of features. One example is the neuron shown at $J9$ between epochs 100 and 500. At epoch 100, it is noisy, then gets dark before turning black, and finally finds a niche and converges to a feature.

Several things can be learned from training just this one model. Firstly, $K = 20$ could be a little high, as many neurons take similar niches, and several do not find any. Secondly, 1000 epochs are more than enough. Most neurons are done learning somewhere between epochs 250 and 500, and even after 250 epochs, the model would be a good classifier. Later experiments justify this statement. Further, the training itself is pretty slow. A higher learning rate or smaller batch size could increase the convergence rate. A higher learning rate will make the updates larger, while a smaller batch size means there are more of them per epoch. This was the object of the following experiment.

8.1.1 Blue Wall of Death

Setting the learning rate to $\eta_0 = 7.5 \cdot 10^{-4}$ and the batch size to 1000, I trained a model with the same parameters as before for 50 epochs. For visibility, the convolution width was also changed to $w = 4$. Figure 8.3 shows the synapses for some of the first and, the last epoch. Immediately it is clear something is very wrong. While many neurons show clear features already after only two epochs, all of these are completely blue two epochs later. After 50 epochs, it looks exactly the same as at epoch 4. The average relative change in the value of the synapses between epochs 2 and 4 was 983%, while between epochs 4 and 50, it was 40%. Almost no learning happened in these 46 epochs, and the model can be considered dead. The quick learning seemed to have broken the winner-takes-all mechanism, causing a phenomenon I dubbed the Blue Wall of Death (BWD). Performing more experiments, it quickly became apparent this phenomenon was quite prevalent and a behaviour needing to be understood and avoided.

The key to understanding BWD was studying the values of the weights of the model. In figure 8.4, I show one single neuron ($B1$ from the model in figure 8.3) at various stages of training. I also show each of the three colour channels separately. In the left-most images, with all channels combined, each image is scaled such that the smallest value is 0 and the largest 1. The colourbar does not apply to these, only the separated channels. For the separate channels, the weights are random at epoch 0. At epoch 2, they are ordered, and one can see the feature detected by the neuron in each channel. At epoch 4, all the values are close to the same negative number. The values of any feature neuron in the model shown in figure 8.1 are all positive. Further study showed that when the anti-Hebbian learning parameter Δ and the weight updates are too high, a single neuron wins all the activation rankings and pushes the other neurons to be all-negative. This was the cause of the BWD. The one winner-neuron can

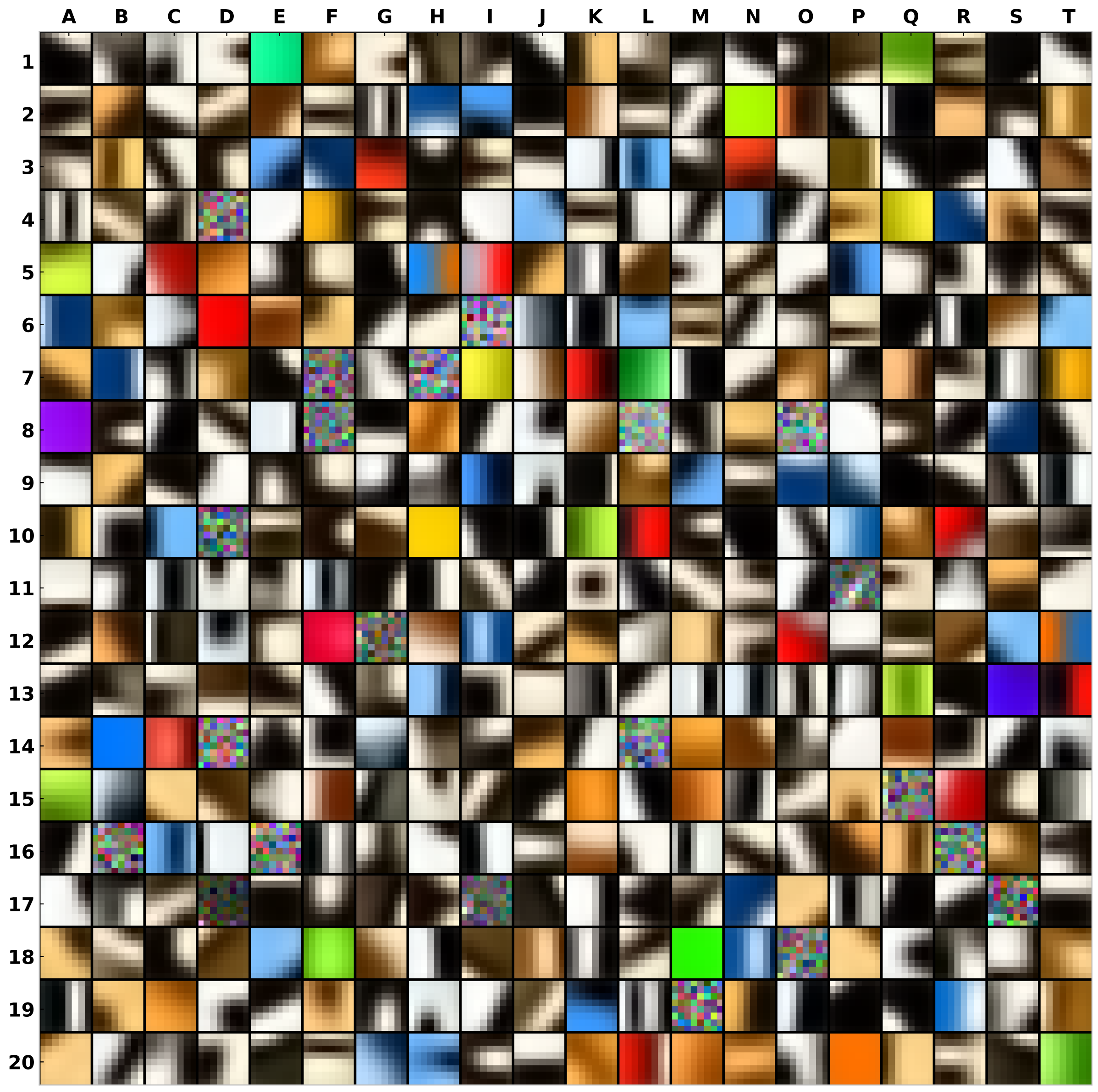


Figure 8.1: Visualisation of the synapses of the trained convolutional layer, with $K^2 = 20$ neurons arranged in a square grid. Each neuron has a kernel width of $w = 8$. The learning rate was $\eta_0 = 1 \cdot 10^{-4}$.

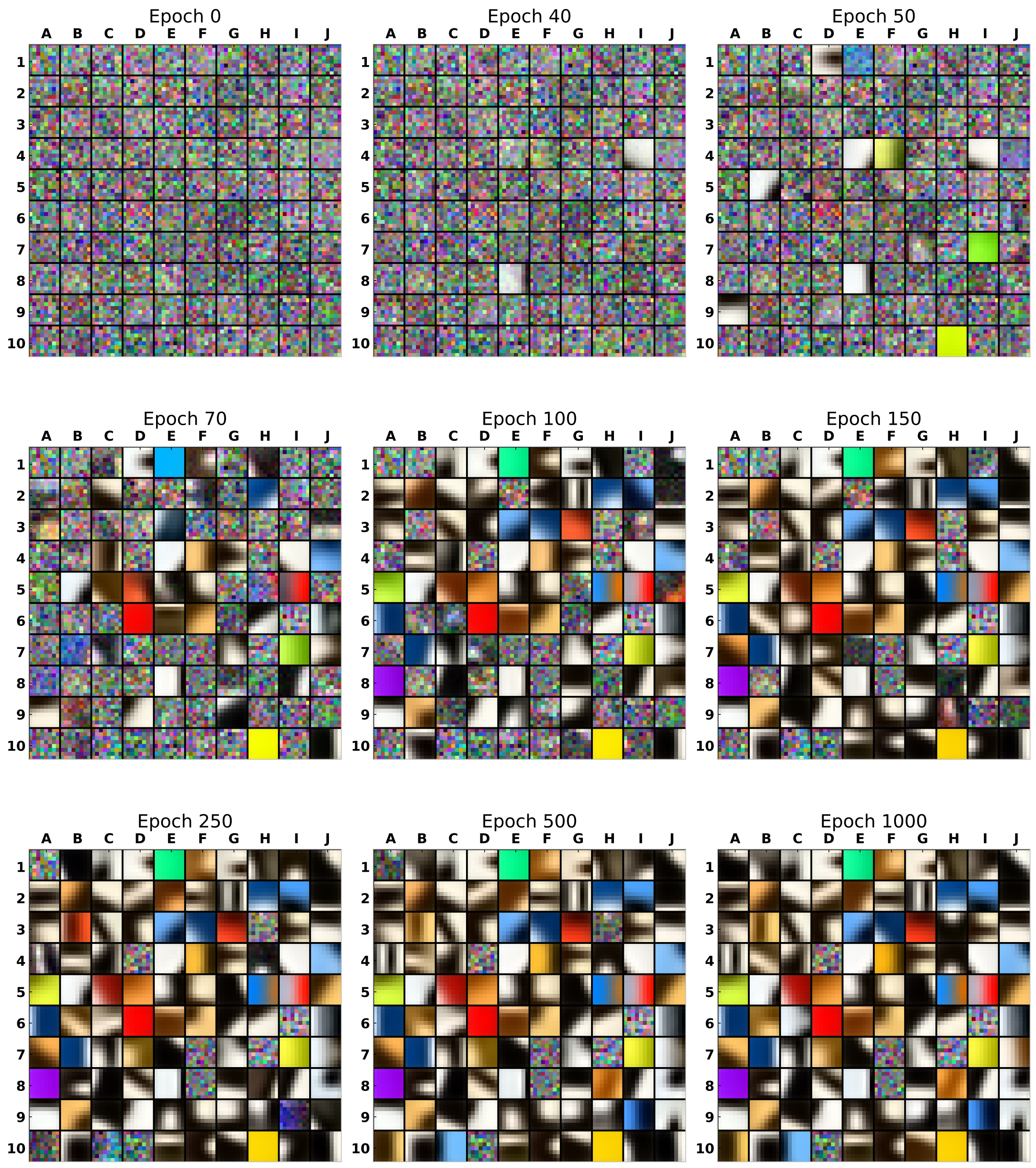


Figure 8.2: Evolution of synapses from figure 8.1 at various times during training.

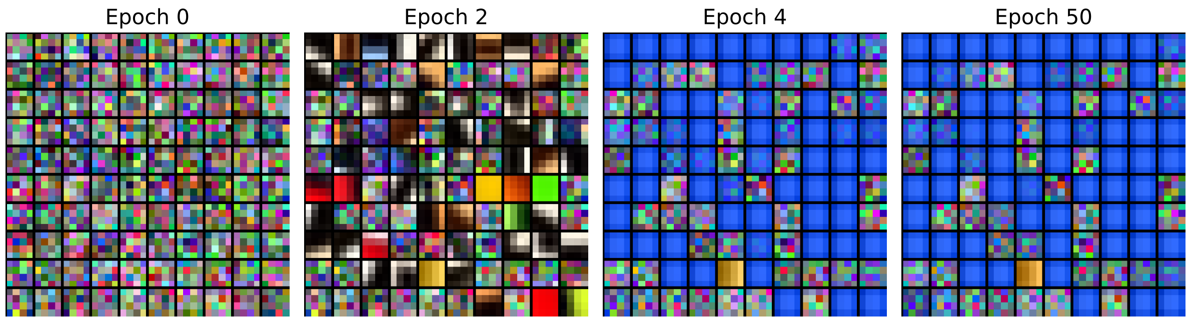


Figure 8.3: Visualised synapses during training using higher learning rate and smaller batch size to speed up training. The learning rate was $\eta_0 = 7.5 \cdot 10^{-4}$, and the batch size was 1000. After four epochs, a significant number of neurons have turned blue, a phenomenon I call the Blue Wall of Death.

be seen at position $E9$ in figure 8.3. In figure 8.5, I show the training of the same model as in figure 8.3, but using $\Delta = 0.15$ instead of $\Delta = 0.2$. Now BWD is avoided. To reduce the number of models with BWD, I started using an even lower value of Δ after this.

Principle *iii* and *iv* in section 4.3.1 require the neurons to converge to have unit p-norm, and this is built into the training scheme equation (4.2). The neuron shown in figure 8.4 has a norm of 1.000 at both epochs 2 and 4, meaning it has converged. I define a feature neuron to consist of all-positive values and have unit-norm, up to a tolerance $1.0 \cdot 10^{-6}$. Then, the percentage of feature neurons to all neurons in the model, f, measures how many neurons have converged. This *f-score* was a useful measure of the "goodness" of the convolution layers and the metric by which I rank them.

8.1.2 Grid Search

Next, I performed a grid search to find parameter sets that yielded convolution layers with high f-ratios, focusing on the learning rate η_0, and anti-Hebbian Δ. I trained model for each combination of the following parameters: $\Delta \in \{0.2, 0.4\}$, $K \in \{14, 20\}$, $m \in \{2, 3, 4, 5\}$, $p \in \{2, 3, 4, 5\}$ and $\eta_0 \in \{7.5 \cdot 10^{-4}, 1.0 \cdot 10^{-3}, 1.25 \cdot 10^{-3}\}$. For each value of the learning rate, I used 5 different learning rate schedulers; linear, cosine and exponential decay. For the exponential decay, I used the following decay rates: $\gamma \in \{2.5, 3.25, 4.0\}$. For all models, I used convolution width $w = 8$, batch size 1000 and trained for 50 epochs. Of the 960 models trained, 581 exhibit BWD and was excluded from further study. In figure 8.6, a histogram shows the f-ratios of the remaining models. One can see that the models are evenly spread out for $0.15 < f < 0.8$, centred around 0.5. There is a peak around 0.95, as many models have very high f-score, close to the maximum of 1. This indicates that several parameter sets let the neurons converge to feature neurons in the training period used but that some leads to slow learning and require more training.

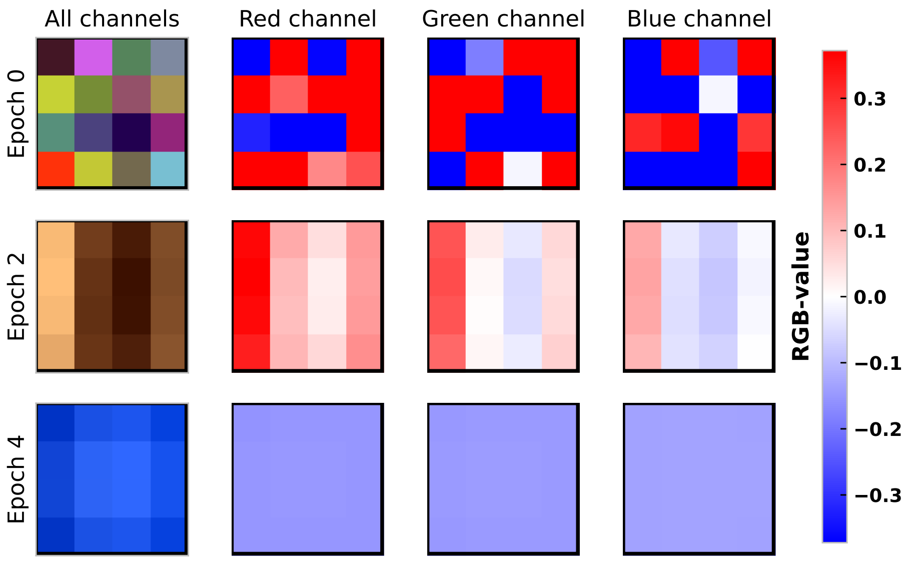


Figure 8.4: Visualised synapses during training using higher learning rate and smaller batch size to speed up training. The learning rate was $\eta_0 = 7.5 \cdot 10^{-4}$, and the batch size was 1000. After four epochs, a significant number of neurons have turned blue, a phenomenon I call the Blue Wall of Death.

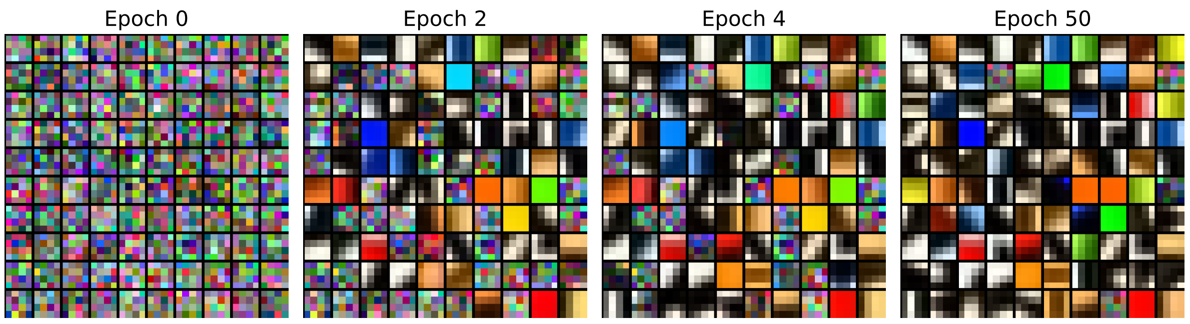


Figure 8.5: A remedy for the Blue Wall of Death-problem. Using a smaller anti-Hebbian learning parameter Δ, 0.15 instead of 0.2, the BWD problem is avoided.

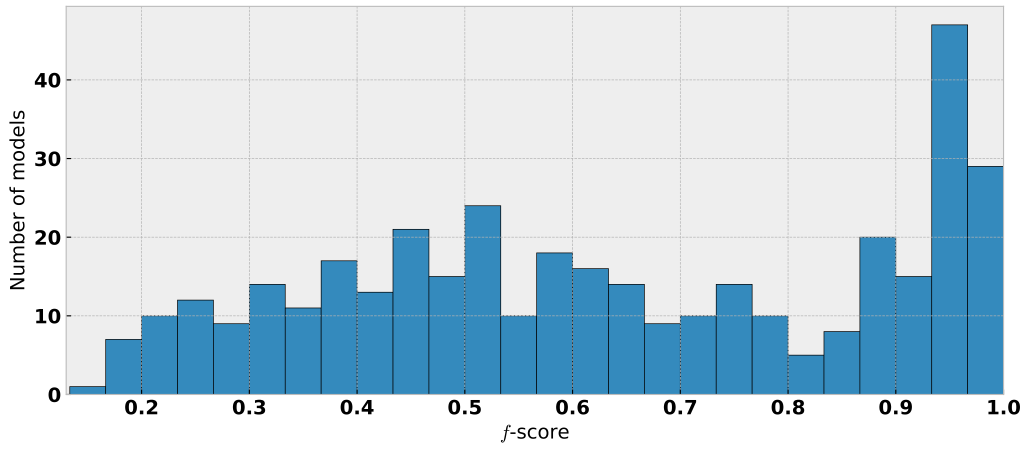


Figure 8.6: A histogram of f-scores of the models from the grid search. Models with $f < 0.15$ are excluded, as they exhibit BWD. This was 581 of the 960 models. Only the remaining 379 models are shown.

To see which hyperparameters were important for getting a high f-score I counted the number of models with the given value of the parameter and with f-score above $0.15, 0.8$ and 0.95. The percentage of models above the thresholds for each value for the parameters is shown in table 8.1. For the anti-Hebbian learning parameter, 76% of all models with $\Delta = 0.2$ had $f > 0.15$, while 11% had $f > 0.95$. However, for $\Delta = 0.4$, only three percent of the models did not get BWD. For K, more models with $K = 20$ got $f > 0.15$ than with $K = 14$, though this is reversed when considering $f > 0.95$. However, the difference is small. For the ranking parameter, $m = 2$ and $m = 3$ give equally many models for all thresholds and much higher than 4 and 5. Using Lebesgue norm $p = 5$ gives the highest percentage of models with $f > 0.15$, but no models with $f > 0.8$. There is not much difference between the different values of η_0, though higher is somewhat better. This is likely because larger synapse updates make it more likely the neurons converge to unit-norm, one of the two criteria for f-score. For the shape of the scheduler, linear and cosine are equally good. The exponentials with different decay rates perform similarly to each other but worse than linear and cosine.

Attempting to get fewer models with BWD, I performed a second grid search. This time, all models used $K = 10$ and $\eta_0 = 1 \cdot 10^{-3}$. The hyperparameters I varied was $\Delta \in \{0.1, 0.2\}$ and $w \in \{4, 8\}$, $m \in \{2, 3, 4, 5, 6\}$ and $p \in \{2, 3, 4, 5, 6\}$. The f-score for all of these models are shown in table A.1 in appendix A. For each convolution layers with $f \geq 0.8$, I trained classifier layers, also as a grid search over the hyperparameters. The values used was $n \in \{2, 4, 6, 8, 10, 12\}$, $\eta_0 \in \{8 \cdot 10^{-4}, 1 \cdot 10^{-3}, 2 \cdot 10^{-2}\}$ and $w_{\text{pool}} \in \{1, 5, 7, 9, 11, 13\}$. For $w_{\text{pool}} \neq 1$, I used $s_{\text{pool}} = 2$, otherwise it was 1. The classifier layers were trained for 100 epochs with a batch size of 1000. The full results are shown in table A.2 in appendix A. Presented here is the number of models with each value of the given parameter, the average test accuracy and the largest test accuracy. The best model used $w = 4$, $\Delta = 0.1$, $m = 6$, $p = 4$, $\eta_0 = 2 \cdot 10^{-3}$, $n = 12$ and $w_{\text{pool}} = 7$, which correctly classified 68.9% of the test data. This is not particularly good, though, due to the constraint on the architecture, it was not expected to be much higher. My result is comparable to the results in [20], which reported an accuracy of 69.5%. Their model used different parameters. The differences can also be explained by variations in the initialisation of the weights. I studied the distribution of models obtained by the same parameters, the result of which is presented in section 9.3.

Table 8.1: Percentage of models with the given value of a hyperparameter and f-score above the given threshold, relative to all models with that value of the hyperparameter.

Parameter	Value	$f > 0.15$	$f > 0.8$	$f > 0.95$
Δ	0.2	76%	26%	11%
	0.4	3%	0%	0%
K	14	35%	12%	7%
	20	44%	14%	4%
m	2	56%	22%	8%
	3	50%	21%	10%
	4	38%	8%	3%
	5	15%	1%	0%
p	2	33%	28%	15%
	3	31%	24%	7%
	4	38%	0%	0%
	5	56%	0%	0%
η_0	$7.5 \cdot 10^{-4}$	39%	12%	3%
	$1.0 \cdot 10^{-3}$	39%	12%	6%
	$1.25 \cdot 10^{-3}$	40%	14%	8%
$\eta(t)$	linear	40%	16%	9%
	cosine	39%	15%	9%
	exp, $\gamma = 2.50$	39%	9%	3%
	exp, $\gamma = 3.25$	40%	13%	3%
	exp, $\gamma = 4.00$	40%	12%	3%

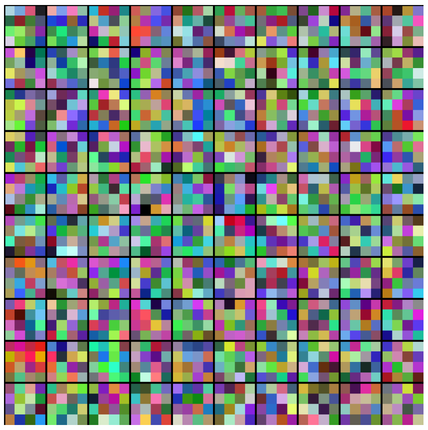
The range of parameters that performed well in the previous grid search was quite large, so I did not fix the value of most of the hyperparameters. However, for all future models, I used the initial learning rate $\eta_0 = 1 \cdot 10^{-3}$ for the convolution layer and $\eta_0 = 2 \cdot 10^{-3}$ for the classifier layer, both with cosine annealing. The convolution layers were trained for 30 epochs, while the classifier was trained for 100 epochs. These numbers were found by looking at where the loss-curve flattened out, and the models did not improve by more training. Finally, attempting to minimise the number of models with BWD, I used $\Delta = 0.1$ for all future models. For the rest of the parameters, I was interested in studying their impact on both the accuracy and robustness, and therefore did not fix them.

A third and final grid search was conducted. Now I used $K \in \{10, 20\}$, $w \in \{4, 8, 12\}$, $m \in \{2, 3, 4, 5, 6\}$ and $p \in \{2, 3, 4, 5, 6\}$ for the convolution layer. 86 of the 150 models had $f \geq 0.8$. For each of these I trained classifier layers using $\sigma \in \{\text{ReLU}, \text{ELU}, \text{GeLU}, \text{rExp}\}$ (equations (2.6) and (2.8) to (2.10)), $w_{\text{pool}} \in \{5, 7, 9, 11, 13\}$ and $n \in \{2, 4, 6, 8, 10, 12\}$. Later this was extended with $n \in \{16, 20, 25, 30, 35, 40, 45, 50\}$ for some, but not all, combinations of the other hyperparameters. $s_{\text{pool}} = 2$ was used for all models. This resulted in 11089 trained models. The results of this grid search are shown in table A.3 in appendix A. The best model used $w = 4$, $K = 20$, $m = 5$, $p = 4$, $w_{\text{pool}} = 9$, $\sigma = \text{ELU}$ and $n = 16$, and achieved an accuracy of 72.2%.

8.2 Training BP-models

It is the group of models from the last grid search I used for determining the robustness. But to have something to compare these to, I needed BP-models. Similar grid searches were done, training models with the backpropagation algorithm. Earlier tests showed that $\eta_0 = 2.0 \cdot 10^{-3}$ with cosine annealing was as good choice of learning rate parameters also for these models. Further, 70 epochs was enough. The grid search used the same values for the relevant hyperparameters as for the LL-models, though $n > 16$ was not used due to bad performance for higher n. The results are shown in table A.4 in appendix A. The model with the highest accuracy used $w = 4$, $K = 20$, $n = 1$, $\sigma = \text{ELU}$ and $w_{\text{pool}} = 9$, and achieved an accuracy of 72.4%. This is slightly better than the best LL-model, getting only 17 more correct classifications on the test dataset, which contains 10000 images in total.

In figure 8.7, I compare the visualisation of the convolution filters of the best BP-model to the best LL-model. For both, only 100 out of the 400 filters are shown. There are clear and obvious differences between the two models. As seen already in figure 8.1, the filters of the LL-models are features like colours, edges or coloured edges. The visualisations of the neurons shown in these figures display the patch that will activate the given neuron the most, and in the LL-models, these patches are easily interpretable. For the BP-models, they all look random. Not much is gained by looking at what patch will activate the neuron the most. However, the weights are not random. To showcase this, I picked the best BP and LL-model using convolution kernel width $w = 12$, meaning they see more of the image at a time and can form more complex filters. Their filters are visualised in figure 8.8. The filters of this LL-model are similar to the one in figure 8.7b but with much smoother edges. While the BP-model still is a black box and not interpretable, there are clear, non-random structures in the filters.


(a) Best BP-model

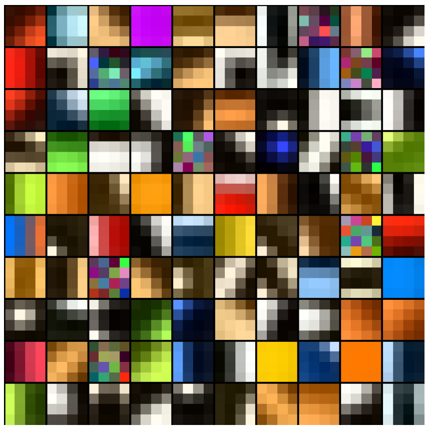
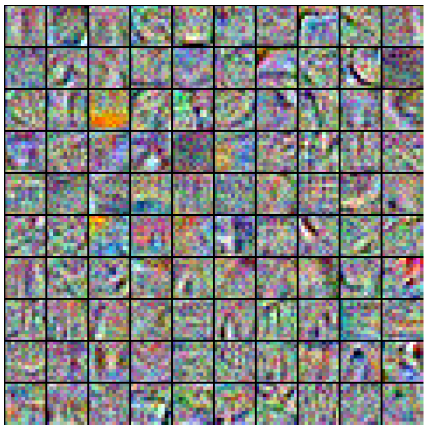

(b) Best LL-model

Figure 8.7: Comparison of the convolution filters in the best BP and LL-model. Only one quarter of the total number of filters are shown for both models. Most of the features are easily interpretable for the LL-model, while for the BP-models all are indistinguishable from random noise. The connection paths from image to class are harder to understand for the BP-models.


(a) Best BP-model using $w = 12$

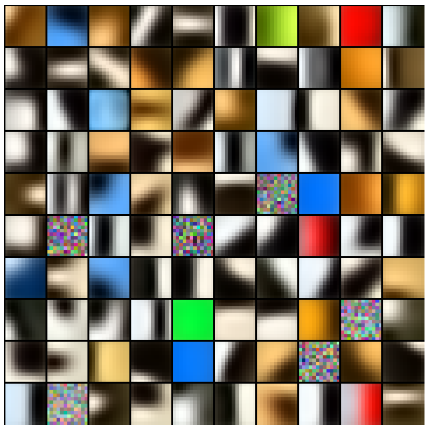

(b) Best LL-model using $w = 12$

Figure 8.8: Comparing the best BP and LL filters that use $w = 12$, which form larger and more complex features. The LL-neurons are smoother but similar to the models with $w = 4$, while the BP-models filters exhibit some structure, clearly not random noise, but are still not at all interpretable.

Chapter 9

Attacking the Models

This chapter presents the results of applying the attacks to the models. I start by showing some example attacks on the best LL-model, then show the results of an attack on the entire test dataset before presenting the results of the attacks on all the models I have trained, using both FGSM and PGD.

9.1 Robustness of LL-models

9.1.1 FGSM

I applied the FGSM attack on the LL-model with the best accuracy from the previous chapter. Four examples of this are shown in figure 9.1. In the left column of images, I show the original, unperturbed image, along with the predicted class and the confidence of the prediction. In the middle image, I show the added perturbation, which for FGSM, is the sign of the gradient of the loss function w.r.t the pixels in the image. The adversarial radius ρ, which is the smallest attack strength for which the model makes a misclassification, is shown above each perturbation. In the right column, I show the final attack image, along with the wrong classification and the prediction confidence. While it is possible to see the difference between the original and attack image for all four examples shown in figure 9.1, the difference is very small. It is easiest to see in the sky behind the ship in image 1 and the bird in image 4. Nonetheless, the model goes from being very certain in the correct classification to completely wrong with the addition of a relatively small perturbation.

The adversarial radius ρ is the smallest attack strength for which the model makes a misclassification. This is determined down to an accuracy of 6 decimal places, meaning $x' = x+(\rho-1\cdot 10^{-6})\cdot\pi$ will not be misclassified by the model, while $x' = x+\rho\cdot\pi$ is. Here π is the perturbation calculated by the FGSM attack. For the same model, I calculated ρ for every image in the dataset for which the model has a confidence of $\mathcal{C} \geq 80\%$ or above. For the best LL-model, there were 4523 images for which the confidence was above this threshold. I plot the accuracy of the model as a function of attack strength ε, in figure 9.2. I also fit equation (7.6) to the curve. The goodness of the fit is determined by the mean squared error (MSE) and the R^2-score. The inflection point of the fitted curve was calculated from equation (7.7). This is the quantity I defined as the robustness of the model. For this model, it was measured to be $\mathcal{R} = 0.01720 \pm 0.00001$. The value does not say much on its own, but it must be compared to the robustness of other models. As seen in the figure, the MSE is very low, and the R^2-score is very close to 1, so this value of $\mathcal{R}$ seems reasonable. This is also reflected in the uncertainty of the value, which was very low, at 0.06%.

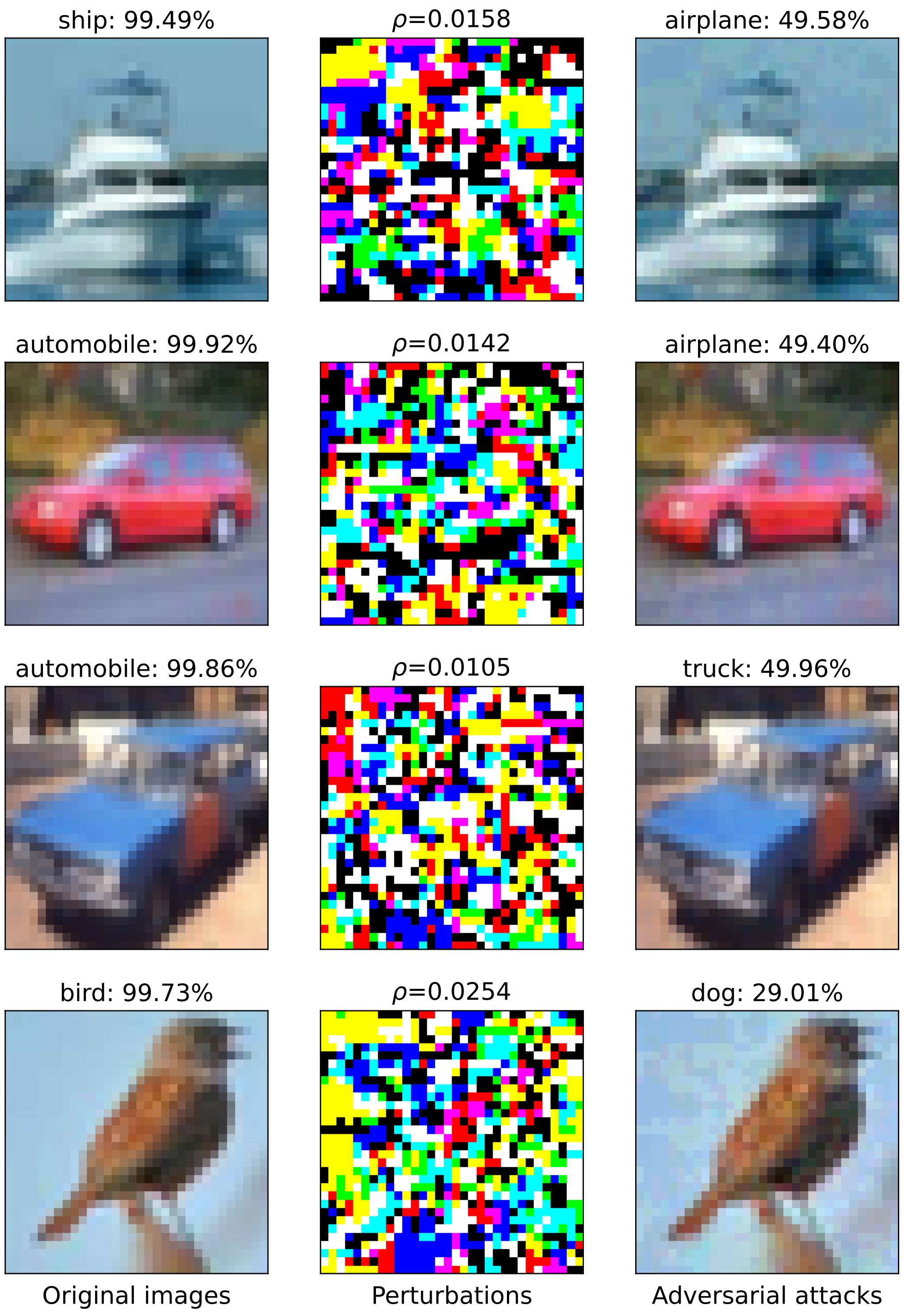


Figure 9.1: 4 examples of FGSM attacks on the best LL-model from the last chapter. The left column shows the original images, the middle column shows the sign of the gradient, and the right column shows the perturbed images. The confidence in the predictions and the attack strength are shown above each image. These are the smallest attack strength for which the model misclassifies the attack image.

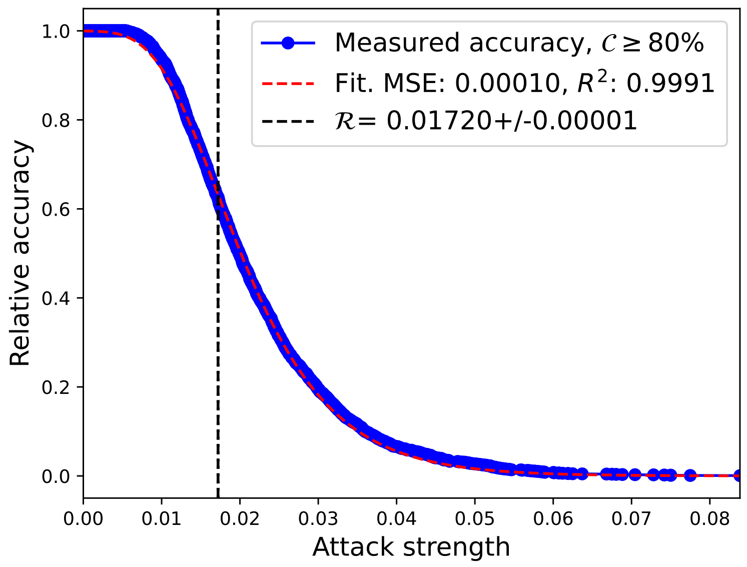


Figure 9.2: Adversarial radius of each image in the dataset, shown as the accuracy of the model as function of attack strength ε. The inflection point of the fitted curve is the robustness of the model. Relative accuracy means that only a subset of the images were used, where the prediction confidence of the model was above 80%, such that when the attack strength is 0, the accuracy is 1

To compare the robustness of the different models, I calculate $\mathcal{R}$ for all the LL-models from the third grid search. I only selected the models where more than 500 images in the dataset were correctly predicted with confidence above $\mathcal{C} = 80\%$. This is to ensure that the robustness is not calculated from a small subset of the dataset. A fit was made of equation (7.6) to the accuracy/attack strength curve of all the models. In the cases where I was not able to obtain a fit with R^2-score above 0.95, I dropped the model from further analysis. In some cases, I was not able to obtain a fit at all, in which case I also dropped the model. This left 8690 out of the 11089 models for which I was able to obtain a value for $\mathcal{R}$. In figure 9.3, I show test accuracy plotted against $\mathcal{R}$ for each of these models. The accuracies of the model range between just below 50%, to just above 70%, and the robustness from $\mathcal{R} = 0.007$ to $\mathcal{R} = 0.047$. However, there is a clear trade-off between accurate and robust models. In the figure, I show the straight line going from the most accurate to the most robust model. Only a very few models are above this line, while many are close. The histograms lining the sides show the number of models in intervals of $\mathcal{R}$ and test accuracy. This shows that the density of models located in the top left corner of the figure decreases steadily going right, towards more robust. The spread of accuracies is broader, with the histogram almost having a bell-shaped curve, though with an extra peak.

In figure 9.3, one can see that most of the models follow the same mode, lying in a banana-shaped band from high accuracy to high robustness, with some outliers on both sides. In many cases, these outliers correspond to specific values of the hyperparameters. To see this and the structure within the banana, I plot the models again for each hyperparameter, where I colour the models by the different values of the parameters. This is shown in figure 9.4. Subfigure **A** show the different activation functions. rExp gives to worst models in both accuracy and robustness, while the other activation functions are more or less the same. However, GeLU seems to make models slightly more robust. Subfigure **B** show that $K = 20$ gives visibly more

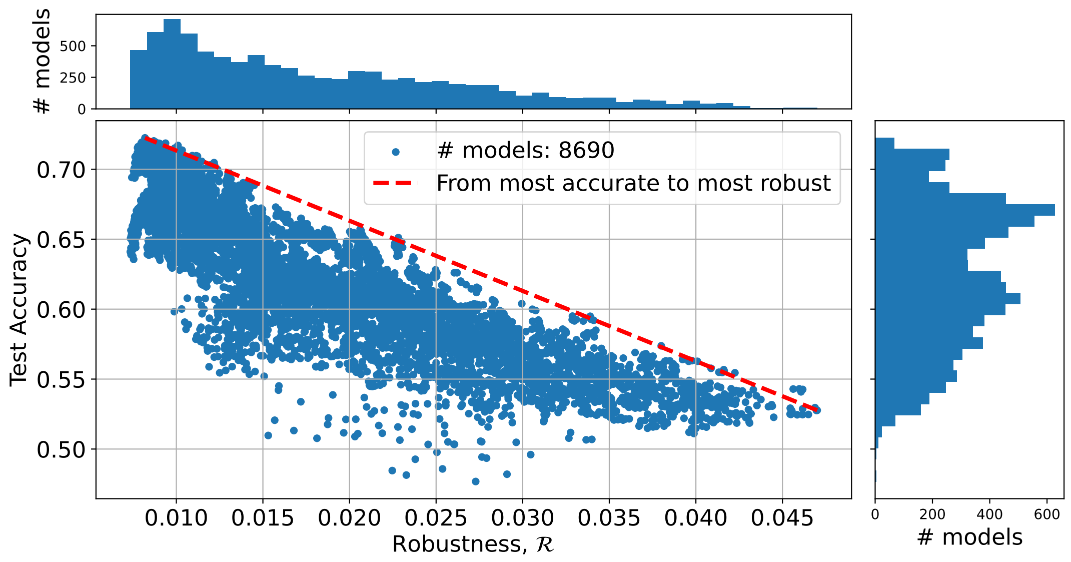


Figure 9.3: The test accuracy vs robustness, shown for all the LL-models from the third grid search. It demonstrates a significant trade-off between accurate and robust models. The straight line is the line going from the most accurate to the most robust model. Very few models are above this line.

accurate models than $K = 10$. Subfigure **C** shows that all the most accurate and least robust models use $w = 4$, while $w = 12$ gives robust but less accurate models. $w = 8$ gives models in between these. Subfigure **D** shows that the two largest Max-Pooling kernel sizes give worse accuracy, but the three smallest values are very similar. One can distinguish the models with different values of K, as there are two bands for each value of w_{pool}. Subfigure **D** and **E** show m and p. These are both hard to interpret, though it can seem that higher m leads to higher accuracy, and lower p makes them more robust. Finally, subfigure **G** and **H** both show the power of the activation function, with the smaller values in **G** and the larger in **H**. The different values of n form narrow, vertical bands. The models with the highest robustness have powers $n \in \{6, 8, 10, 12, 20\}$. In **G**, the models with $n = 0$ correspond to those with $\sigma = \text{rExp}$, all performing very poorly.

The specifics of how accuracy and robustness depended on the values of some of the hyperparameters is a bit hard to interpret from figure 9.4. The number of hidden neurons clearly is important for the accuracy, and within this, the Max-Pooling further separate the models vertically. The kernel width and the power are important for robustness. The ranking parameter m does not visibly impact either the accuracy or the robustness. It is unclear how the p-norm does. While most robust models have $p = 2$ or $p = 3$, these are also the most common values of p in the dataset. I only selected these values when training more models to save time, which makes it possible that other values of p would have performed similarly.

Two important parameters deciding which models I would train and attack were the f-score of the convolution layer and the prediction confidence $\mathcal{C}$, which again affected the number of images the robustness was calculated from. In figure 9.5, I show the accuracy/robustness trade-off, but with each model coloured by f-score (left) and the number of samples used (right). There is great variability in the f-scores, but some trends can be observed. The most accurate models have a low f-score. I had one hypothesis that some of the filters that did not converge were used to convey extra information and caused the models to be more eas-

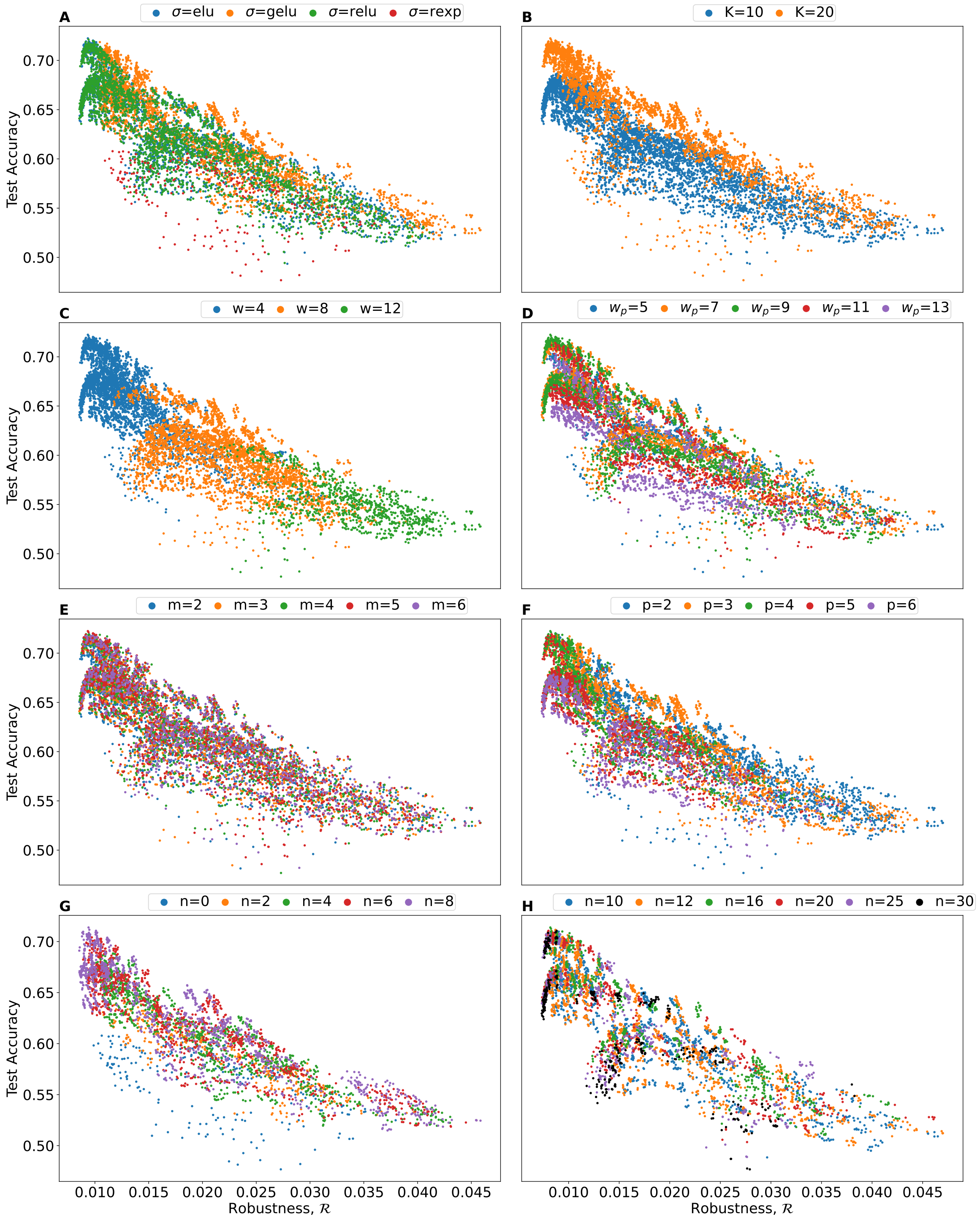


Figure 9.4: The accuracy/robustness plot from figure 9.3, but now coloured by the different values of the hyperparameters. The different plots correspond to the different hyperparameters. Figure **A** shows the different activation functions, **B** shows K, **C** w, **D** w_{pool}, **E** m, **F** p, and **G** and **H** both show the power, for different values of n.

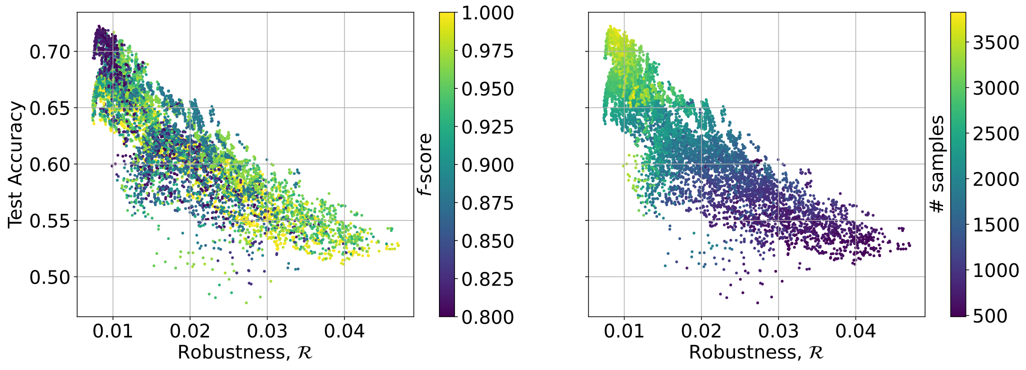


Figure 9.5: The accuracy/robustness trade-off, with each model coloured by the f-score of the convolution layer (left), and the number of samples the robustness was calculated from (right). The most accurate models have a low f-score, while the most robust have a high f-score. The most accurate models are confident about many models, while the highest robustnesses are calculated from the fewest samples.

ily fooled due to the less smooth filters. This was supported by the very high f-score of the most robust models. However, this hypothesis was tested buy comparing the average activity of each neuron across many images and shown to not be the case. The average activity of unconverged neurons was much smaller than converged neurons.

The right graph in figure 9.5 shows that the robustness of the most accurate models was based on many more images. The most robust models barely passed the $>$ 500-images threshold. This was concerning, as the robustness factor I defined was supposed to be a measure of the adversarial radius of the entire dataset. One way of increasing the number of images the robustness is calculated from is to decrease the confidence threshold. I performed another FGSM attack on all the models, now with $\mathcal{C} = 60\%$. Going too low would change the shape of the robustness curve too much. The results are shown in figure 9.6, together with the previous results. More models are included, and the shape of the superstructure is preserved. It can appear that it is rotated around the most accurate models. However, this is not the case, as the attack only measures the robustness. Therefore, the models have simply moved to the left, and more have been added. The decrease in robustness is most pronounced for the most robust models, while some models on the far left actually seem to have become more robust. This causes the superstructure to appear rotated.

9.1.2 PGD

The PGD algorithm generates even stronger attack images than FGSM, so I also applied this attack method. This method takes two parameters, the projection size S and step length ε. It also measures two things, the adversarial radius ρ and the number of steps needed to reach it. Using $S = 0.14$ and $\varepsilon = 5 \cdot 10^{-4}$, I attack the model with the hyperparamterers $K = 20$, $w = 4$, $p = 4$, $m = 5$, $w_{\text{pool}} = 9$, $n = 12$ and $\sigma = \text{ELU}$, and plot the robustness curve in figure 9.7. The granularity comes from the fixed step size $\varepsilon = 5 \cdot 10^{-4}$. I also show the number of steps used in the graph on the right. As seen in the figure, the curves have the exact same shape. The robustness is determined in the same way as for FGSM, but now also done for the number of steps. This gives two different measures of the robustness of a model. However, notice that $\mathcal{R}_\varepsilon = \varepsilon \mathcal{R}_N$. This relation was found to be true for all models and for different values of PGD parameters S and ε. This is because I measure distance in the L_∞-norm and use the

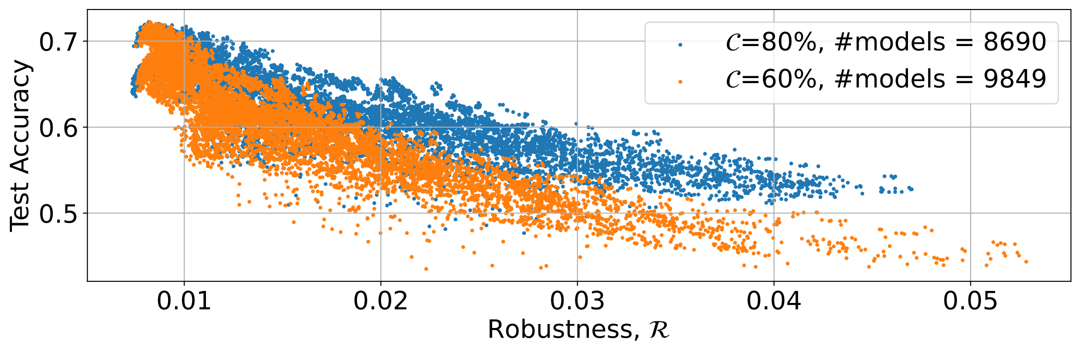


Figure 9.6: Test accuracy vs robustness, with FGSM attack performed using both $\mathcal{C} = 60\%$ and $\mathcal{C} = 80\%$. The lower threshold of $\mathcal{C}$ lets more models have more than 500 samples, making the tail longer. However, the models are less robust as the average confidence is lower, which shifts the models to the left in the figure. The most robust models are most affected by the inclusion of more images. This gives the appearance of a rotation of the superstructure. However, this is not the case.

gradient sign in the perturbation update at each iteration. It is sufficient that the loss points in the same direction at every step in a single dimension for this relation to be found. With the dimensionality of the images being $32 \cdot 32 \cdot 3 = 3072$ and the typical number of steps taken before misclassification being well below 100, this is likely to be the case. For simplicity, I will therefore only consider $\mathcal{R} = \mathcal{R}_\varepsilon$ going forward.

Using both $S = 0.12$ with $\varepsilon = 1 \cdot 10^{-3}$ and $S = 0.14$ with $\varepsilon = 5 \cdot 10^{-5}$, I apply the PGD attack on all the models and plot the accuracy/robustness, together with the results from the FGSM attack, in figure 9.8. As expected, PGD is a stronger attack, resulting in lower robustness than FGSM. As in figure 9.6, where $\mathcal{C} = 60\%$ decreased the robustness of more robust models more, PGD also affect the most robust models the most, though the effect is smaller. Further, $S = 0.14$ resulted in lower robustness than $S = 0.12$. The difference in the two measures was the same for all models.

9.2 Comparing LL and BP-models

Now I compare the robustness of the LL-models to the BP-models. In the previous section, I showed that while the robustness of a single model differed for FGSM and PGD and depended on the value of $\mathcal{C}$ and the PGD parameters, the relative robustness between models was similar. I therefore do not need to apply all the attacks with the same values to compare the LL-models and the BP-models. I still show that the behaviour of the robustness of the BP-models is similar for the two attack methods, in figure 9.9. In general, the robustness of the LL-models is higher than the BP-models. However, for the most accurate models, the robustness is very similar. While the LL-models span a wider range of robustnesses, the BP-models span a wider range of accuracies. The same filtering criteria were applied to both types of classifiers. However, the classifier layer for LL-models was only trained if the convolution layer was reasonably good, determined by the f-score of the filters. This means there was an extra quality assurance the LL-models had to pass, making the worst LL-models more accurate than the worst BP-models.

Further, while PGD results in lower robustness for both LL-models and BP-models than FGSM, the decrease is the largest for the LL-models. While the BP models barely move to the left

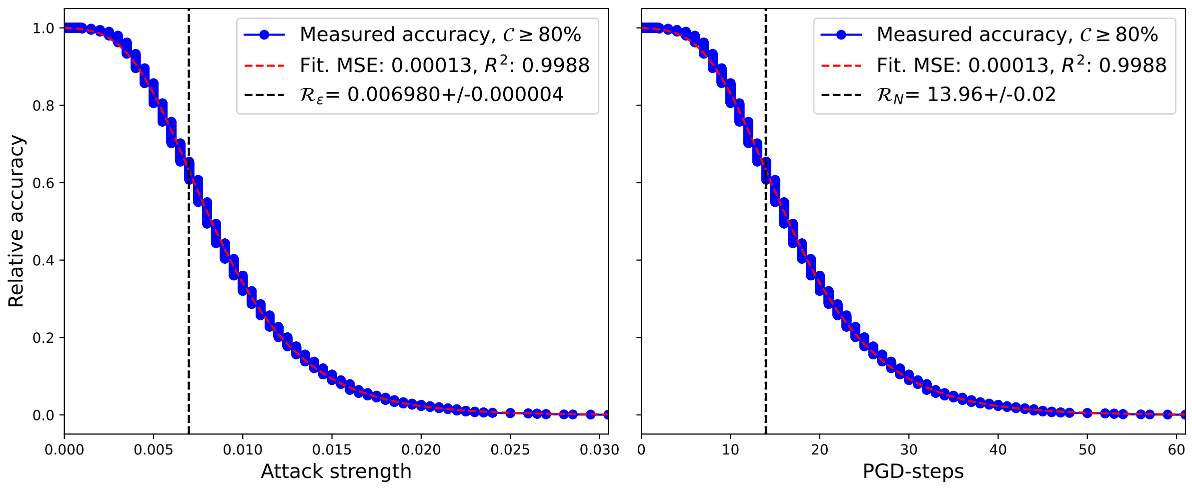


Figure 9.7: The robustness curve for a model for PGD. The granularity in the attack strength is due to the step length parameter ε in the method, which here was set to $\varepsilon = 5 \cdot 10^{-4}$. The number of steps needed to reach the adversarial radius is shown in the right graph. These two curves have the same shape, and the robustnesses determined from them are related by $\mathcal{R}_\varepsilon = \varepsilon \mathcal{R}_N$.

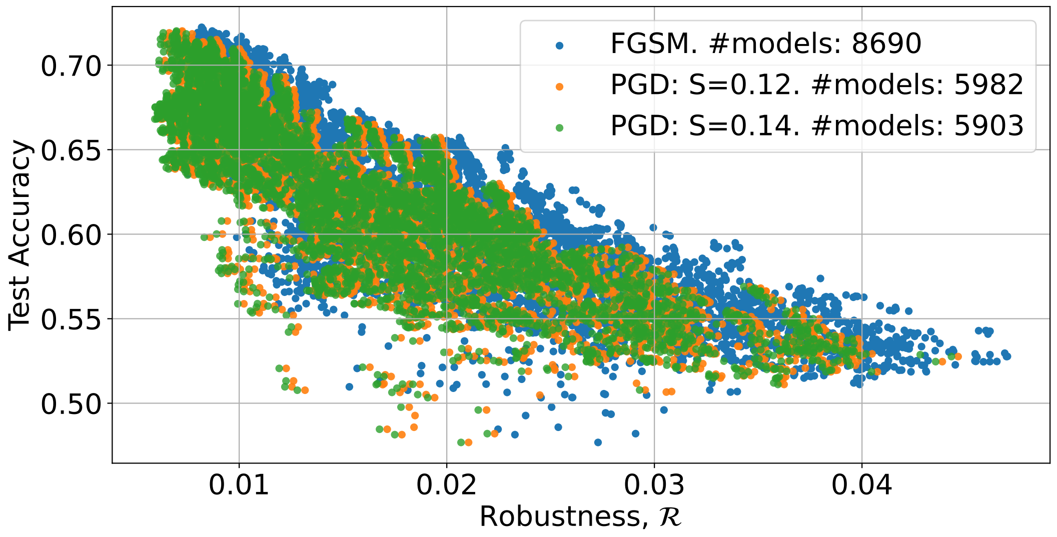


Figure 9.8: The accuracy/robustness trade-off for PGD with two different sets of parameters, along with FGSM. More models are excluded by the 500-image limit in the PGD attacks, but all yield similar structures. The PGD gives a lower robustness, with $S = 0.14$ slightly lower than $S = 0.12$.

when comparing figure 9.9b to figure 9.9a, the LL models take a big step left. Interestingly, the very least robust BP models actually have higher robustness against PGD than FGSM. I believe this is due to the lower resolution in the adversarial radius ρ obtained by PGD due to the fixed step size ε, though I did not have enough time to study if this was the exact cause.

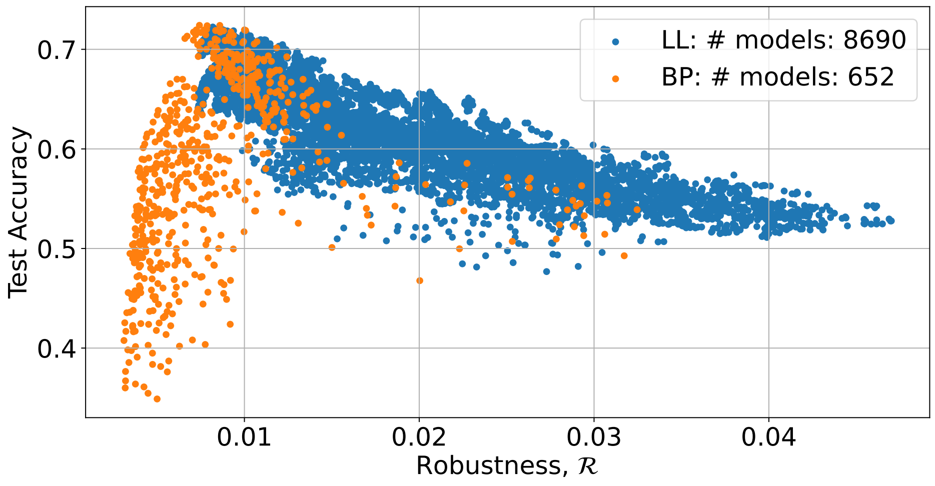
LL-models lie in a single band with some outliers, while the BP-models fall into three distinct regions. The first one has the highest accuracy, with comparable robustness to the best LL-models. The second region is a column of models with very low robustness and spanning many accuracies. There is a significant gap between these two regions. A third region is a group of models with moderate accuracies but much higher robustness than the other BP-models. This region spans a large area, though it is very sparely populated. A study equivalent to the one presented in figure 9.4 was performed to identify the hyperparameters responsible for each of the three regions. For most of the hyperparameters, the effect of the hyperparameters was the same as the effect on the LL-models. However, the power n is not typically used for BP-models, and worth mentioning. The first region is populated by models with $n = 1, 2$, which is as expected. Models with $n = 2$ performed worse in both metrics compared to $n = 1$ models, though not by much. Curiously, the top region also contain the models trained with the rExp activation function. The third region is populated by models using ELU, and $n = 10, 12, 16$. The higher the n, the higher the robustness. This was unexpected, as there is no precedence for using steep activation functions together with backpropagation and getting such high robustness is interesting, especially since the models here are more robust and more accurate than the models with the same value of n from the second region. The second region consists of the rest of the models, with the accuracies decreasing with increasing n. The gap between the first and second regions would likely have been populated by models with $n = 3$.

9.3 Ensembles of models

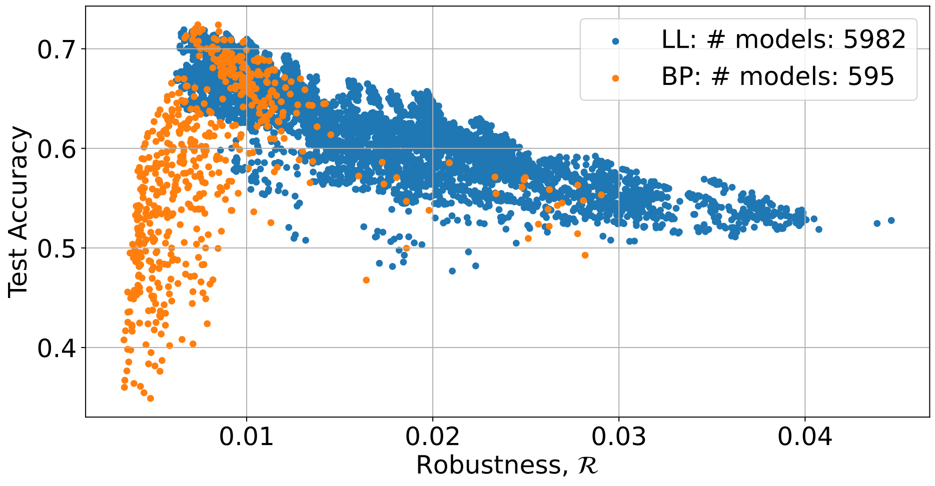
In the previous sections, I only trained and attacked a single model for each unique set of hyperparameters. However, due to the random initialisation of weights, a set of hyperparameters will result in a different value of accuracy and robustness if used several times. In this chapter, I will explore this distribution.

I trained ten convolution layers of the LL-models for each of the sets of parameters $K \in \{10, 20\}$, $w \in \{4, 12\}$, $m \in \{3, 4\}$, $p \in \{2, 4\}$. None of the 160 layers had BWD, and 119 had f-score above 0.8. Only these were used further. For each convolution layer, I trained ten classifier layers for each set of the parameters $n \in \{4, 12\}$ and $w_{\text{pool}} \in \{5, 9\}$. $\sigma = \text{GeLU}$ was used for all models. For the BP-models, I trained ten models for each set of the parameters $K \in \{10, 20\}$, $w \in \{4, 12\}$, $n \in \{4, 12\}$, $\sigma \in \{\text{ELU}, \text{GeLU}, \text{rExp}\}$ and $w_{\text{pool}} \in \{5, 9, 11\}$. These sets of parameters were not chosen to maximise the robustness or accuracy, merely to study their variances. Due to limited time, I did not vary the activation function for the LL-models, and used more Max-pooling kernel sizes for the BP-models. BP-models are significantly quicker to train. Further, because I trained ten classifiers for each convolution layer, of which there already were ten for each set of parameters, I had 100 LL-models per set of parameters. There were only ten BP-models per set.

I applied both the FGSM and PGD attack on the ensembles of models. The accuracy/robustness of the FGSM attacks is shown in figure 9.10. The models are clustered and coloured by the set of parameters, and the mean and standard deviation of the robustness and accuracy is shown for each cluster. As can be seen, the spread is quite small for most of the clusters. The exception is some of the more robust clusters to the right in the figures. The average accuracy,



(a) FGSM



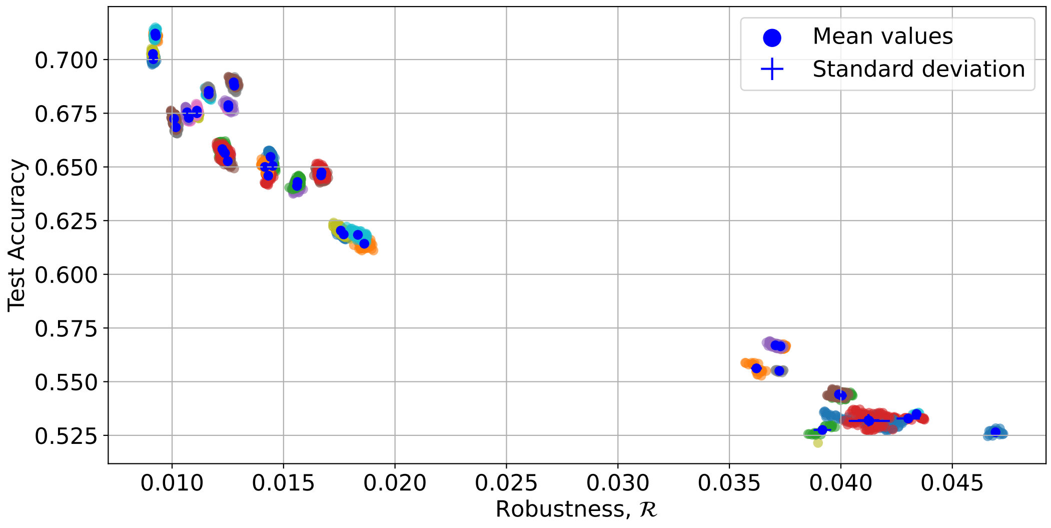
(b) PGD

Figure 9.9: Accuracy/robustness for LL and BP-models using FGSM (Top) and PGD (Bottom). On the whole, the LL-models are more robust than the BP-models. However, for the most accurate models, the robustness is similar.

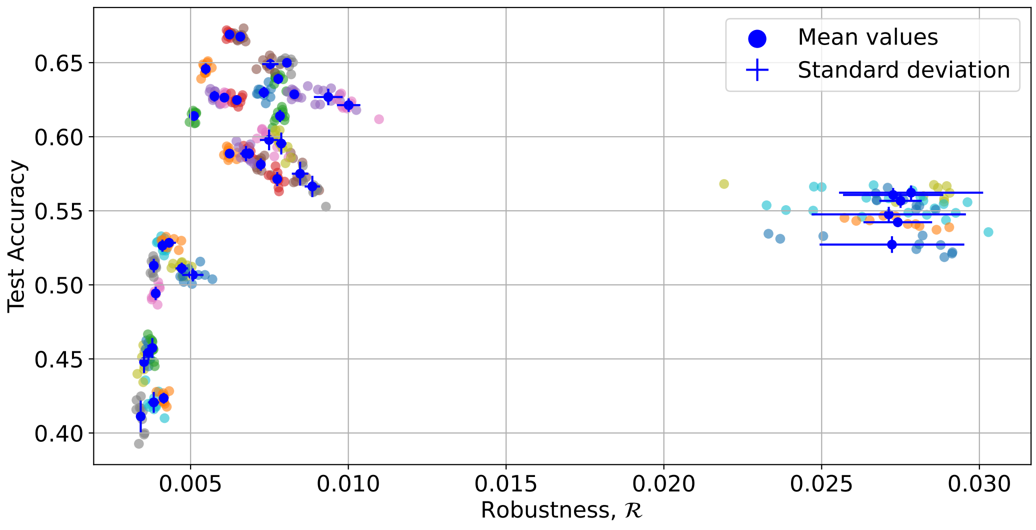
Table 9.1

	LL-models	BP-models
Accuracy	$(0.21 \pm 0.09)\%$	$(0.9 \pm 0.4)\%$
Robustness (FGSM)	$(0.7 \pm 0.3)\%$	$(0.8 \pm 0.7)\%$
Robustness (PGD)	$(3 \pm 2)\%$	$(3 \pm 2)\%$

robustness, and standard deviations for the FGSM attack applied on the LL-models are shown in table B.1 in appendix B. The values for the PGD attack on the LL-models are shown in table B.2. The results for the BP-model are in table B.3 and table B.4, for FGSM and PGD attack, respectively. While the average accuracy and robustness vary significantly, as seen in figure 9.10, the spread within each group was relatively small. In table 9.1, I show the relative spread of the uncertainties in the tables in appendix B. The LL-models have a much lower spread in accuracy than the BP-models; $(0.21 \pm 0.09)\%$ compared to $(0.9 \pm 0.4)\%$. This can, at least in part, be explained by the sharing of the convolution layer in the LL-models. Every BP-model has unique convolution filters, contributing to the higher variance. Training 100 BP models for each set could also have given lower variance. Further, while the spread in robustness against PGD is the same, the LL-models have a much lower variance in the spread of robustness against FGSM.



(a) FGSM on ensembles of LL-modes



(b) FGSM on ensembles of BP-modes

Figure 9.10: Attacks on ensembles of models

Chapter 10

Discussion

10.1 Training and selecting models

In this thesis, I aimed to study the robustness properties of convolutional neural networks trained with the Krotov-Hopfield learning rule. To do this, I trained many models with different hyperparameters to be able to study how these affected the robustness of the models. In addition, I trained models with the backpropagation algorithm to compare the robustness of the two training algorithms. While I was able to show that the LL models are more robust than the BP models in a single figure, figure 9.9a, the story behind why the models ended up where they did in terms of accuracy and robustness is more complicated.

From the beginning, I was interested in studying how the various hyperparameters would affect the accuracy and robustness of the models. Using similar values as was presented in [19] and [20], I trained many models with different hyperparameters. This made me stumble upon the Blue Wall of Death, where the winner-takes-all mechanism would break down and have a single neuron win all the time, causing all other neurons to turn blue. While reducing the anti-Hebbian learning parameter Δ mitigated the problem, it was never completely avoided, which made me develop the f-score to determine the goodness of the convolution layers. In the rest of my work, I only considered layers with $f \geq 0.8$. This threshold was set arbitrarily and perhaps too high, as was later seen. Many models that achieved high accuracy were barely above this limit. If a similar metric is used to filter the models in future work, I would recommend using a lower threshold, especially for models with many hidden nodes, where the f-score is more likely to be low.

I did not explore the use of pruning on the convolution layer. Pruning is a standard machine learning technique and involves removing little used nodes. This results in fewer parameters and computations during inference and can also be used to determine particularly important nodes. I would have used it to remove the unconverged neurons, giving all models an f-score of 1. Pruning can also improve the generalisation if the model overfitted to the training data, increasing test accuracy. I did study the average activity of the unconverged neurons and found this to be much lower than the other neurons. However, how these contribute to the classification, and thus accuracy and robustness was not studied.

The method I adopted instead of pruning was essentially pruning entire models with too low f-score. This was done to save time, as the training of many models was a slow process. One effect of this was that I did not have a model of all values of the hyperparameters in the first grid searches. While many of the models with $0.2 < f < 0.8$ likely would not have been very good, having their accuracy would have made it easier to compare and determine the effect of each value of the hyperparameters. I attempted to do this in section 9.3, with the ensembles of models. This let me isolate the random variation of the initialisation. Further, I performed a more limited grid search, only comparing two values for each hyperparameter. For p, I used 2 and 4. Studying the table B.1 in appendix B, I can conclude that $p = 4$ gives significantly more accurate but significantly less robust models than $p = 2$, as this was the only variable changed. However, I conducted this study too late for it to inform the earlier parts of the thesis. For instance, $m = 3$ did not give a notable change in accuracy or robustness compared to $m = 4$, which would have let me exclude the hyperparameter m from the grid search.

One limitation of the ensemble test was that I did not record the f-scores of the various models, which would have let me conclude how this is correlated with the accuracy and robustness. This is corroborated by the kernel width w results in table B.1. Only $w = 4$ consistently gave me models to attack, while many rows with $w = 12$ are missing due to no models passing the various quality assurances. Comparing these where possible reveals that $w = 12$ gives much more robust models, though less accurate. Looking at the number of models in each case of $K = 10$ and $w = 12$ reveals that likely only 2-3 of the ten convolution layers passed the f-score limit, further underlining that I set the threshold too high. However, for $K = 20$, there are many more models with $w = 12$. $K = 10$ gives less accurate and more robust models than $K = 20$. For n, it is $n = 4$ that gives lower accuracy and higher robustness compared to $n = 12$. While these parameters have been simple to compare, always one value with higher accuracy and lower robustness, this is not true for the max-pooling kernel size. $w_{\text{pool}} = 5$ always gives higher accuracy than $w_{\text{pool}} = 9$ for $K = 10$, though this is flipped for $K = 20$. The robustness of $w_{\text{pool}} = 9$ when $K = 20$ is always higher, but for $K = 10$, there is no trend in which value gives more robust models.

Regarding the BP models in table B.3, $K = 20$ always gave higher robustness and lower accuracy than $K = 10$. As for the LL models, the max-pooling kernel width for the BP models behaves strangely. Of the three values I used, there is no trend in which gives the most or least accurate models, while for the robustness, larger w_{pool} is always better. I did not compare the activation function for the LL models, but for BP, ELU always gave more accurate and more robust models than GeLU. Lastly, the kernel width and the power are closely linked. $w = 4$ gave higher accuracy when $n = 4$, and lower when $n = 12$. The robustness was lower for $w = 4$ when $n = 4$ and higher when $n = 12$. $n = 4$ gave both higher accuracy and higher robustness was higher when $w = 4$, though again this trend was reversed when $w = 12$.

One curious observation I made was that the BP models using large n were significantly more robust in some cases. This was only seen when the kernel size was $w = 12$. Neither [19] nor [20] used the hyperparameter n for the BP-model they compared their LL-models to. This is because the power comes from the higher-order energy function of the DAM associated with the LL-models. There is, therefore, no motivation to use it for the BP-models. However, I tried and found that it gave higher robustness. Whether this can be used to make other BP models more robust is left for future work.

While the various hyperparameter affects the robustness and accuracy in different ways, the one thing that repeats throughout the previous paragraphs is that there is a trade-off between high accuracy and high robustness. This is exhibited between the individual hyperparameters and, consequently, among all the trained models. It is apparent in all the accuracy/robustness figures shown in the previous chapter. This trade-off is the main finding of this thesis.

Such a trade-off between accuracy and robustness is a known feature of neural networks [52, 53, 54]. It is usually seen when techniques have been applied to increase the robustness, such as adversarial training or regularisation. Adversarial training is including adversarial attacks in the training data. It is a very computation-heavy process because it requires computing the attacks at every epoch. Regularisation is a technique to prevent overfitting and is usually done by adding a term to the loss function. The convergence to unit p-norm of the synapses required in the KH-rule (equation (4.7)) is a form of regularisation. Regularisation works by preventing some weights from getting too large. My results are consistent with this, as higher p-norm in the KH-model promotes larger synapses, and I found that $p = 2$ gave higher robustness than $p = 4$. This thesis did not thoroughly investigate why the other hyperparameters give a similar accuracy/robustness trade-off.

I included the activation function rExp in my search because of the equivalence between DAMs and FFNNs with activation functions of order n. Higher n gives higher memory capacity in DAMs, and the idea was that the exponential function is a polynomial of order ∞. However, this is not true, so there is no reason why it should work well as an activation function. My finding was that it did not. LL-models with rExp were less accurate and less robust than all the other LL-models.

10.2 Attacking the Models

The robustness against attacks of the Krotov-Hopfield model was already tested in [20] and found to be more robust than similar models trained with backpropagation. However, they only used white-box attacks, such as random noise, occlusion and partial shadows. Further, they only attacked the best model they found. The contributions of this work were to test the models against first-order adversaries, like FGSM, and the universal first-order adversary, PGD [2]. I also found the robustness of many models with different parameters and accuracies. As already stated, I found there to be a trade-off between accuracy and robustness, which is explained by the various choices of hyperparameters. I also studied the effect of changing the parameters that determined the robustness.

I devised the method I used to measure the robustness quantitatively. There is no standard method for defining it in the literature. Robustness is a complex phenomenon which a single value can not capture. Most authors leave it as a qualitative measure and only show the accuracy as a function of the attack strength [2]. These are curves similar to those I showed in figures 9.2 and 9.7. I wanted to define it quantitatively to perform direct comparisons between models and also compare many models at a time. While I was able to do this and showed that the robustness of the models behaved similarly between different ways of measuring it, my measure failed to address many complexities surrounding robustness. For instance, I had to select only images for which the model had prediction confidence above a threshold $\mathcal{C}$ before the logistic shape of the curve appeared. Even when only using correctly classified images, the curve had the shape of exponential decay. While robustness is supposed to measure the ability to withstand adversarial attacks on the entire test dataset, in practice, only a fraction of all the images was used in my experiments. For the results presented in figure 9.3, where $\mathcal{C} = 80\%$, the model with the most samples used 3826, which is barely more than half of those it predicted correctly. This is one of the reasons I set the hard limit of at least 500 images to be included in the comparisons. The other reason was to be able to get a good fit, though if this had been the single reason, the limit could have been lower.

Another limitation of my method was that the function I fit the curve assumed the accuracy would go to 0 for large attack strengths. While there are some images the models genuinely did not misclassify, no matter how strong the FGSM attack became, the larger issue was with PGD, where I project the images back into a region close to the original image. Because I did not use random restarts, only starting from the original image and tracing a single path, there are many images that the models did not misclassify after a high number of iterations. In these cases, I dropped the image from further analysis. Thus, the right asymptote of the accuracy curve being 0 is not strictly true, as it only counts the images which were misclassified at some point. This problem of my method is the opposite of that discussed above. It exacerbates the issue of my quantitative measure of robustness, not considering the entire dataset but from the other side.

A deeper issue with my method was revealed by the relation $\mathcal{R}_\varepsilon = \varepsilon\mathcal{R}_N$, shown in figure 9.7 and found to hold for all of the models. It means that in at least one pixel-dimension, the gradient of loss points in the same direction at every iteration of PGD. As already stated, this is indeed very likely, due to the dimensionality of the space and the number of iterations typically taken. The implication of this is that the PGD attack was not necessarily the strongest it could have been. The difference between IFGSM and how I defined PGD was the projection operation, and I think I likely used too large values for the projection radius S. $S = 0.1$, the lowest I tried, is more than twice as high as the largest robustness measure I got from FGSM. While some images certainly were "projected" by the attack, most were not, and the adversarial radius on these images would be the same as for FGSM. However, S is the largest adversarial radius that can be measured, and a lower value would not give a logistic-shaped accuracy curve. The choice of the inflection point of a logistic curve as the robustness was a decision based on the observed behaviour of many accuracy/attach-strength curves, not a deeper philosophical insight into what the robustness of a neural network means.

The issue presented in the previous paragraph is only a potential issue. It assumes that inside the smaller projection radius S, there is a misclassified point. [10] studied the decision boundaries in the pixel-space for backpropagation-trained models and showed them to be very messy when no regularisation technique was used. However, with regularisation, they become much smoother. The KH-rule has built-in regularisation, though of a different kind from those studied in [10]. I did not have time to study how the decision boundaries look in the models I trained. If they are as smooth as obtained by regularisation in [10], it is entirely likely that the high projection radius S I used for PGD was reasonable. Further study is needed to determine this.

Chapter 11

Conclusions & Future Work

The main question I set out to answer in this work was whether the Krotov-Hopfield local learning rule for unsupervised training of one layer of a neural network would be more robust against adversarial attacks compared to models trained with the backpropagation algorithm. Robustness is the ability of a model to resist adversarial attacks, where imperceptible perturbations to images aim to fool the model. To compare the robustnesses of many models, I defined a quantitative measure of the robustness, something most authors do not attempt due to the complexities involved in giving exact definitions of "robustness". I trained a large number of models, exploring the effects of the various hyperparameters. The best performing models had accuracies of 72%, both for the backpropagation and local learning-trained models. While their robustnesses were similar, I found that when comparing the robustness of all the models I trained, the LL models where much more robust. I also found that the LL models had a trade-off between accuracy and robustness, while the BP models did not.

While there are several issues with my method for determining the robustness, as was discussed in the previous chapter, I was able to do what I set out to do and got a value for the robustness of each of the models I trained. This let me compare them and determine exactly how the different hyperparameters affected the robustness and accuracy. Therefore, I consider my method a success, but one which needs to be seen in the light of these issues. I further find it encouraging that the behaviour of the robustness on all the models I trained was similar no matter which parameters I used for the attacks. Thus, I can conclude that the LL-models were more robust than BP-models. While individual BP-models can be more robust than individual LL-models, the vast majority of LL-models I trained were more robust than the vast majority of BP-models. All the BP-models with high robustness used unconventional, steep activation functions. More significantly, I found an accuracy/robustness trade-off exhibited in the LL-models, similar to findings in the literature when regularisation methods are applied. This trade-off was less apparent in the BP-models, likely due to no usage of regularisation methods. I also found this trade-off in other hyperparameters, not just regularisation.

I also wanted to look at the interpretability of these models. Machine learning models usually learn patterns in a way that can not be understood from the outside. I showed that this was true for the BP-models by visualising the convolution filters, which looked like they consisted of random colours. However, the LL-models have had much smoother filters, displaying single colours or sharp edges. While not all the filters had converged to these nice features, I found that these did not have a high average activation. I also found that the models with the highest robustness had a smaller fraction of these unconverged filters.

11.1 Outlook

The origin of the Krotov-Hopfield rule that I used limits the architecture of the neural network to one hidden layer. This severely restricts the capability and predictive power of the models. The highest accuracy I was able to obtain in any model was 72% correct predictions. This has been shown to improve slightly by using several convolution layers with different kernel widths in parallel, and the robustness of such a model is yet to be determined. I found that the kernel width is very important for the robustness, so it is possible that the robustness of a model with several kernel widths would be higher than what was found in this work and perhaps even not come at the cost of accuracy. This conjecture remains to be tested.

Another extension of the architecture could be achieved by breaking the connection to DAMs, and using the Krotov-Hopfield rule to train many layers of deep neural networks. While the training process is more inefficient than backpropagation, the higher robustness and built-in regularisation could make it worth it.

By training an ensemble of models for many sets of hyperparameters, I isolated the effect of each of these on the accuracy and robustness. Most resulted in a trade-off between accurate and robust models. While the norm parameter p does this by restricting large weights, the mechanism for why the other does was not studied.

I found that models trained with backpropagation and which used unconventional activation functions, ELU to the power of n, where $n = 10, 12$ and 16, resulted in significantly higher robustness. If this is a general result, it could be used to improve the robustness of models trained with backpropagation, though it might also come at the cost of accuracy.

Part V

Appendices

Appendix A

Grid search results

This appendix contains the result of the various grid searches.

Table A.1 shows the f-score for each set of the hyperparameters $\Delta \in \{0.1, 0.2\}$, $w \in \{4, 8\}$, $p \in \{2, 3, 4, 5\}$, and $m \in \{2, 3, 4, 5\}$. All models used $K = 10$. For each of the models in table A.1 with $f > 0.8$, I performed a grid search of classification layers, which let me test the accuracy of the models on the test dataset. Table A.2 shows the number of models, average accuracy and best accuracy for each of the values of the hyperparameters $\Delta \in \{0.1, 0.2\}$, $w \in \{4, 8\}$, $p \in \{2, 3, 4, 5\}$, $m \in \{2, 3, 4, 5\}$, $\eta_0 \in \{8.0 \cdot 10^{-4}, 1.0 \cdot 10^{-3}, 2.0 \cdot 10^{-3}\}$ and $n \in \{2, 4, 6, 8, 10, 12\}$. From these results, I fixed the learning rate parameters for both the convolution and classifier layers. All future convolution layers was trained with $\eta_0 = 0.001$, while the classifier layers used $\eta_0 = 0.002$, both with cosine annealing. Wanting to see the effect on both accuracy and robustness, I did not fix the other parameters.

A final, wider grid search was conducted, using $w \in \{4, 8, 12\}$, $K \in \{10, 20\}$, $p \in \{2, 3, 4, 5, 6\}$, $m \in \{2, 3, 4, 5, 6\}$, $n \in \{2, 4, 6, 8, 10, 12, 16, 20, 25, 30, 35, 40, 45, 50\}$, $\sigma \in \{\mathrm{ReLU}, \mathrm{ELU}, \mathrm{GeLU}, \mathrm{rExp}\}$ and $w_{\mathrm{pool}} \in \{5, 7, 9, 11, 13\}$. The higher values of n were added later, and only used for the best-performing values of the other parameters. Table A.3 shows the number of models for each parametes, along with average and best accuracy, where the highest value for each hyperparameter is highlighted in bold. The best model was found to be the one with $w = 4$, $K = 20$, $p = 4$, $m = 5$, $n = 16$, $\sigma = \mathrm{ELU}$ and $w_{\mathrm{pool}} = 9$, with an accuracy of 72.2%. However, all of the models were used in the later robustness tests.

Table A.1: Results of the second grid search. Four hyperparameters were varied, and the models were compared by the f-score. The vast majority of the models either had BWD (and consequently $f \approx 0$), or had $f \geq 0.8$. The threshold of 0.8 was set arbitrarily to save time when training.

Δ	w	p	m	f-score	Δ	w	p	m	f-score	Δ	w	p	m	f-score
0.1	4	2	2	0.99	0.1	4	2	3	0.96	0.1	4	2	4	0.95
0.1	4	2	5	0.91	0.1	4	2	6	0.04	0.1	4	3	2	0.94
0.1	4	3	3	0.92	0.1	4	3	4	0.88	0.1	4	3	5	0.89
0.1	4	3	6	0.05	0.1	4	4	2	0.93	0.1	4	4	3	0.88
0.1	4	4	4	0.82	0.1	4	4	5	0.84	0.1	4	4	6	0.80
0.1	4	5	2	0.97	0.1	4	5	3	0.94	0.1	4	5	4	0.85
0.1	4	5	5	0.75	0.1	4	5	6	0.74	0.1	4	6	2	0.98
0.1	4	6	3	1.00	0.1	4	6	4	0.96	0.1	4	6	5	0.87
0.1	4	6	6	0.81	0.1	8	2	2	1.00	0.1	8	2	3	1.00
0.1	8	2	4	1.00	0.1	8	2	5	1.00	0.1	8	2	6	0.05
0.1	8	3	2	0.97	0.1	8	3	3	0.98	0.1	8	3	4	0.97
0.1	8	3	5	0.95	0.1	8	3	6	0.94	0.1	8	4	2	0.87
0.1	8	4	3	0.92	0.1	8	4	4	0.89	0.1	8	4	5	0.87
0.1	8	4	6	0.83	0.1	8	5	2	0.85	0.1	8	5	3	0.84
0.1	8	5	4	0.89	0.1	8	5	5	0.86	0.1	8	5	6	0.86
0.1	8	6	2	0.89	0.1	8	6	3	0.86	0.1	8	6	4	0.88
0.1	8	6	5	0.89	0.1	8	6	6	0.85	0.2	4	2	2	1.00
0.2	4	2	3	0.02	0.2	4	2	4	0.03	0.2	4	2	5	0.03
0.2	4	2	6	0.03	0.2	4	3	2	0.94	0.2	4	3	3	0.02
0.2	4	3	4	0.03	0.2	4	3	5	0.04	0.2	4	3	6	0.05
0.2	4	4	2	0.95	0.2	4	4	3	0.02	0.2	4	4	4	0.03
0.2	4	4	5	0.04	0.2	4	4	6	0.05	0.2	4	5	2	0.95
0.2	4	5	3	0.76	0.2	4	5	4	0.14	0.2	4	5	5	0.04
0.2	4	5	6	0.05	0.2	4	6	2	0.96	0.2	4	6	3	0.77
0.2	4	6	4	0.24	0.2	4	6	5	0.22	0.2	4	6	6	0.05
0.2	8	2	2	0.99	0.2	8	2	3	0.02	0.2	8	2	4	0.03
0.2	8	2	5	0.03	0.2	8	2	6	0.03	0.2	8	3	2	0.97
0.2	8	3	3	0.96	0.2	8	3	4	0.02	0.2	8	3	5	0.03
0.2	8	3	6	0.04	0.2	8	4	2	0.88	0.2	8	4	3	0.79
0.2	8	4	4	0.64	0.2	8	4	5	0.04	0.2	8	4	6	0.05
0.2	8	5	2	0.81	0.2	8	5	3	0.76	0.2	8	5	4	0.57
0.2	8	5	5	0.04	0.2	8	5	6	0.05	0.2	8	6	2	0.80
0.2	8	6	3	0.79	0.2	8	6	4	0.63	0.2	8	6	5	0.40
0.2	8	6	6	0.05	-	-	-	-	-	-	-	-	-	-

Table A.2: Results of the first grid search with classifier layers. The number of models with each value of the given parameter is shown, along with the average and best accuracy, measured on the test dataset. The results for the values of w_{pool} are not included due to missing data. The best value for each parameter, both overall and on average, is shown in bold.

Parameter	Value	#models	Avg. accuracy	Max accuracy
Δ	0.1	1705	**58.0**%	**68.9**%
	0.2	424	57.2%	67.0%
w	4	1034	**60.1**%	**68.9**%
	8	1095	55.6%	60.4%
m	2	744	57.2%	67.1%
	3	401	57.2%	66.7%
	4	385	58.2%	68.5%
	5	360	58.3%	68.7%
	6	239	**59.0**%	**68.9**%
p	2	370	54.4%	65.8%
	3	467	57.3%	68.4%
	4	476	58.9%	**68.9**%
	5	360	58.4%	66.9%
	6	456	**59.5**%	67.9%
η_0	$8.0 \cdot 10^{-4}$	674	57.3%	67.0%
	$1.0 \cdot 10^{-3}$	671	57.4%	67.0%
	$2.0 \cdot 10^{-3}$	756	**58.8**%	**68.9**%
n	2	343	50.8%	61.9%
	4	344	56.1%	65.5%
	6	342	58.2%	66.2%
	8	371	59.8%	68.2%
	10	364	60.4%	68.7%
	12	365	**60.8**%	**68.9**%

Table A.3: Results of the final grid search. The number of models with each value of the given parameter is shown, along with the average and best accuracy, measured on the test dataset. The best value for each parameter, both overall and on average, is shown in bold.

Parameter	Value	#models	Avg. accuracy	Max accuracy
w	4	4662	**64.2**%	**72.2**%
	8	4408	57.0%	67.6%
	12	2016	50.0%	61.1%
K	10	7306	57.3%	68.7%
	20	3780	**61.7**%	**72.2**%
m	2	2393	**59.1**%	71.9%
	3	2267	58.9%	71.8%
	4	2268	59.0%	71.9%
	5	2394	58.7%	**72.2**%
	6	1764	57.9%	71.8%
p	2	3653	56.3%	71.9%
	3	3024	58.9%	71.9%
	4	1763	**62.1**%	**72.2**%
	5	1386	60.7%	71.9%
	6	1260	58.6%	67.8%
n	2.0	528	56.2%	65.3%
	4.0	1320	59.1%	68.8%
	6.0	1320	60.9%	70.4%
	8.0	1320	61.7%	71.4%
	10.0	1319	62.1%	71.9%
	12.0	1320	62.3%	72.0%
	16.0	792	**63.2**%	**72.2**%
	20.0	527	62.5%	71.9%
	25.0	528	61.2%	71.7%
	30.0	528	59.4%	71.9%
	35.0	176	46.5%	60.4%
	40.0	176	40.6%	54.5%
	45.0	176	35.5%	48.2%
	50.0	176	31.2%	42.3%
$\sigma(x)$	ELU	3168	**61.7**%	**72.2**%
	GeLU	3871	56.1%	71.9%
	ReLU	3167	61.6%	72.1%
	rExp	440	52.4%	60.8%
w_{pool}	5	2023	**60.8**%	70.6%
	7	3168	58.6%	71.7%
	9	2903	58.0%	**72.2**%
	11	1496	59.2%	71.4%
	13	1496	57.4%	70.3%

Table A.4: Results of the grid search with BP-models The number of models with each value of the given parameter is shown, along with the average and best accuracy, measured on the test dataset. The best value for each parameter, both overall and on average, is shown in bold.

Parameter	Value	#models	Avg. accuracy	Max accuracy
w	4	260	**57.4**%	**72.4**%
	8	260	55.7%	72.0%
	12	260	48.7%	70.5%
K	10	390	52.8%	70.2%
	20	390	**55.0**%	**72.4**%
n	1.0	90	**68.6**%	**72.4**%
	2.0	90	66.9%	70.9%
	4.0	90	62.0%	67.0%
	6.0	90	56.9%	63.4%
	8.0	90	51.4%	60.1%
	10.0	90	44.9%	58.6%
	12.0	90	40.1%	57.1%
	16.0	90	33.9%	55.3%
σ	ELU	240	59.8%	**72.4**%
	GeLU	240	48.0%	**72.4**%
	ReLU	240	51.4%	72.0%
	rExp	30	**66.6**%	71.5%
w_{pool}	5	156	53.9%	71.7%
	7	156	**54.6**%	72.0%
	9	156	54.5%	**72.4**%
	11	156	53.9%	**72.4**%
	13	156	52.8%	71.5%

Appendix B

Ensembles of models

Table B.1: Result of FGSM attack on the ensembles of LL-models: the number of models successfully attacked for each set of parameters, along with the average test accuracy and robustness within the group.

K	w	m	p	n	w_{pool}	# models	Accuracies	$\mathcal{R}$
10	4	3	2	4	5	100	0.619 ± 0.001	0.0177 ± 0.0001
10	4	3	2	4	9	100	0.614 ± 0.001	0.0186 ± 0.0002
10	4	3	2	12	5	100	0.650 ± 0.001	0.01451 ± 0.00005
10	4	3	2	12	9	100	0.646 ± 0.002	0.01432 ± 0.00007
10	4	3	4	4	5	100	0.656 ± 0.002	0.0124 ± 0.0001
10	4	3	4	4	9	100	0.653 ± 0.002	0.0125 ± 0.0001
10	4	3	4	12	5	100	0.673 ± 0.001	0.01075 ± 0.00006
10	4	3	4	12	9	100	0.668 ± 0.002	0.01017 ± 0.00007
10	4	4	2	4	5	100	0.620 ± 0.002	0.0176 ± 0.0001
10	4	4	2	4	9	100	0.618 ± 0.001	0.0183 ± 0.0002
10	4	4	2	12	5	100	0.655 ± 0.002	0.01442 ± 0.00007
10	4	4	2	12	9	100	0.650 ± 0.002	0.0142 ± 0.0001
10	4	4	4	4	5	100	0.658 ± 0.002	0.0123 ± 0.0001
10	4	4	4	4	9	100	0.657 ± 0.002	0.0123 ± 0.0002
10	4	4	4	12	5	100	0.675 ± 0.002	0.01067 ± 0.00004
10	4	4	4	12	9	100	0.672 ± 0.002	0.01009 ± 0.00008
10	12	3	4	4	5	10	0.5550 ± 0.0004	0.0372 ± 0.0001
10	12	3	4	12	5	10	0.5350 ± 0.0003	0.04339 ± 0.00009
10	12	4	2	12	5	22	0.526 ± 0.001	0.0469 ± 0.0002
10	12	4	4	4	5	23	0.556 ± 0.002	0.0362 ± 0.0002
10	12	4	4	4	9	20	0.528 ± 0.002	0.0392 ± 0.0004
10	12	4	4	12	5	30	0.5328 ± 0.0003	0.0430 ± 0.0005
20	4	3	2	4	5	100	0.641 ± 0.001	0.01561 ± 0.00009
20	4	3	2	4	9	100	0.646 ± 0.002	0.0167 ± 0.0001
20	4	3	2	12	5	100	0.6778 ± 0.0009	0.01252 ± 0.00007
20	4	3	2	12	9	100	0.688 ± 0.002	0.01279 ± 0.00008
20	4	3	4	4	5	100	0.6750 ± 0.0009	0.01112 ± 0.00005
20	4	3	4	4	9	100	0.684 ± 0.001	0.01164 ± 0.00007
20	4	3	4	12	5	100	0.7002 ± 0.0009	0.00916 ± 0.00005
20	4	3	4	12	9	100	0.711 ± 0.001	0.00927 ± 0.00005
20	4	4	2	4	5	100	0.643 ± 0.002	0.01563 ± 0.00009
20	4	4	2	4	9	100	0.647 ± 0.002	0.0167 ± 0.0001
20	4	4	2	12	5	100	0.679 ± 0.001	0.01252 ± 0.00009
20	4	4	2	12	9	100	0.689 ± 0.001	0.0128 ± 0.0001
20	4	4	4	4	5	70	0.676 ± 0.001	0.01111 ± 0.00004
20	4	4	4	4	9	70	0.685 ± 0.001	0.01164 ± 0.00006
20	4	4	4	12	5	70	0.703 ± 0.001	0.00914 ± 0.00004
20	4	4	4	12	9	70	0.712 ± 0.001	0.00925 ± 0.00004
20	12	3	2	4	5	97	0.532 ± 0.002	0.0413 ± 0.0009
20	12	3	2	12	5	100	0.5665 ± 0.0006	0.0373 ± 0.0001
20	12	3	2	12	9	101	0.5435 ± 0.0008	0.0400 ± 0.0002
20	12	4	2	4	5	77	0.532 ± 0.003	0.0412 ± 0.0005
20	12	4	2	12	5	100	0.5669 ± 0.0007	0.0371 ± 0.0002
20	12	4	2	12	9	98	0.544 ± 0.001	0.0399 ± 0.0002

Table B.2: Result of PGD attack on the ensembles of LL-models: the number of models successfully attacked for each set of parameters, along with the average test accuracy and robustness within the group.

K	w	m	p	n	w_{pool}	# models	Accuracies	$\mathcal{R}$
10	4	3	2	4	5	100	0.619 ± 0.001	0.0164 ± 0.0001
10	4	3	2	4	9	100	0.614 ± 0.001	0.0167 ± 0.0001
10	4	3	2	12	5	100	0.650 ± 0.001	0.01371 ± 0.00005
10	4	3	2	12	9	100	0.646 ± 0.002	0.01319 ± 0.00006
10	4	3	4	4	5	100	0.656 ± 0.002	0.01170 ± 0.00009
10	4	3	4	4	9	100	0.653 ± 0.002	0.01157 ± 0.00009
10	4	3	4	12	5	100	0.673 ± 0.001	0.01033 ± 0.00005
10	4	3	4	12	9	100	0.668 ± 0.002	0.00966 ± 0.00006
10	4	4	2	4	5	100	0.620 ± 0.002	0.0163 ± 0.0001
10	4	4	2	4	9	100	0.618 ± 0.001	0.0165 ± 0.0002
10	4	4	2	12	5	100	0.655 ± 0.002	0.01362 ± 0.00006
10	4	4	2	12	9	100	0.650 ± 0.002	0.01305 ± 0.00008
10	4	4	4	4	5	100	0.658 ± 0.002	0.01159 ± 0.00009
10	4	4	4	4	9	100	0.657 ± 0.002	0.0114 ± 0.0001
10	4	4	4	12	5	100	0.675 ± 0.002	0.01027 ± 0.00004
10	4	4	4	12	9	100	0.672 ± 0.002	0.00961 ± 0.00007
10	12	3	4	12	5	10	0.5350 ± 0.0003	0.03923 ± 0.00007
10	12	4	2	12	5	6	0.5264 ± 0.0006	0.0441 ± 0.0004
10	12	4	4	4	5	26	0.556 ± 0.002	0.0336 ± 0.0002
10	12	4	4	4	9	10	0.5257 ± 0.0003	0.0342 ± 0.0001
10	12	4	4	12	5	26	0.5328 ± 0.0003	0.0390 ± 0.0005
20	4	3	2	4	9	93	0.646 ± 0.002	0.0154 ± 0.0001
20	4	3	2	12	5	100	0.6778 ± 0.0009	0.01208 ± 0.00006
20	4	3	2	12	9	100	0.688 ± 0.002	0.01210 ± 0.00007
20	4	3	4	4	5	100	0.6750 ± 0.0009	0.01074 ± 0.00005
20	4	3	4	4	9	100	0.684 ± 0.001	0.01104 ± 0.00006
20	4	3	4	12	5	100	0.7002 ± 0.0009	0.00901 ± 0.00005
20	4	3	4	12	9	100	0.711 ± 0.001	0.00897 ± 0.00004
20	4	4	2	4	5	6	0.641 ± 0.001	0.0147 ± 0.0001
20	4	4	2	4	9	88	0.647 ± 0.002	0.0154 ± 0.0001
20	4	4	2	12	5	100	0.679 ± 0.001	0.01207 ± 0.00008
20	4	4	2	12	9	100	0.689 ± 0.001	0.01208 ± 0.00009
20	4	4	4	4	5	70	0.676 ± 0.001	0.01074 ± 0.00004
20	4	4	4	4	9	70	0.685 ± 0.001	0.01104 ± 0.00005
20	4	4	4	12	5	70	0.703 ± 0.001	0.00900 ± 0.00004
20	4	4	4	12	9	70	0.712 ± 0.001	0.00895 ± 0.00003
20	12	3	2	4	5	89	0.531 ± 0.002	0.039 ± 0.001
20	12	3	2	12	5	100	0.5665 ± 0.0006	0.0353 ± 0.0001
20	12	3	2	12	9	54	0.5433 ± 0.0007	0.0369 ± 0.0002
20	12	4	2	4	5	68	0.532 ± 0.003	0.039 ± 0.001
20	12	4	2	12	5	100	0.5669 ± 0.0007	0.0351 ± 0.0001
20	12	4	2	12	9	42	0.5446 ± 0.0009	0.0368 ± 0.0002

Table B.3: Result of FGSM attack on the ensembles of BP-models: the number of models successfully attacked for each set of parameters, along with the average test accuracy and robustness within the group.

K	w	n	w_{pool}	σ	# models	Accuracies	$\mathcal{R}$
10	4	4	5	elu	10	0.630 ± 0.004	0.0073 ± 0.0002
10	4	4	5	gelu	10	0.589 ± 0.003	0.0062 ± 0.0001
10	4	4	9	elu	10	0.639 ± 0.003	0.0078 ± 0.0001
10	4	4	9	gelu	10	0.589 ± 0.003	0.0069 ± 0.0001
10	4	4	11	elu	10	0.629 ± 0.002	0.0083 ± 0.0001
10	4	4	11	gelu	10	0.581 ± 0.004	0.0072 ± 0.0001
10	4	12	5	elu	10	0.494 ± 0.005	0.0039 ± 0.0001
10	4	12	5	gelu	10	0.41 ± 0.01	0.00342 ± 0.00009
10	4	12	9	elu	10	0.511 ± 0.004	0.0047 ± 0.0002
10	4	12	9	gelu	10	0.421 ± 0.007	0.0038 ± 0.0002
10	4	12	11	elu	10	0.507 ± 0.004	0.0051 ± 0.0003
10	4	12	11	gelu	10	0.423 ± 0.004	0.0041 ± 0.0001
10	12	4	5	elu	10	0.614 ± 0.004	0.0078 ± 0.0001
10	12	4	5	gelu	10	0.571 ± 0.005	0.0077 ± 0.0001
10	12	4	9	elu	10	0.627 ± 0.006	0.0094 ± 0.0005
10	12	4	9	gelu	10	0.575 ± 0.008	0.0085 ± 0.0003
10	12	4	11	elu	10	0.621 ± 0.005	0.0100 ± 0.0004
10	12	4	11	gelu	10	0.566 ± 0.007	0.0089 ± 0.0002
10	12	12	5	elu	9	0.562 ± 0.005	0.028 ± 0.002
10	12	12	9	elu	10	0.548 ± 0.005	0.027 ± 0.002
10	12	12	11	elu	10	0.527 ± 0.006	0.027 ± 0.002
20	4	4	5	elu	10	0.646 ± 0.004	0.00548 ± 0.00009
20	4	4	5	gelu	10	0.614 ± 0.003	0.00511 ± 0.00007
20	4	4	9	elu	10	0.669 ± 0.002	0.00623 ± 0.00007
20	4	4	9	gelu	10	0.627 ± 0.004	0.0058 ± 0.0001
20	4	4	11	elu	10	0.668 ± 0.003	0.0066 ± 0.0001
20	4	4	11	gelu	10	0.626 ± 0.003	0.00607 ± 0.00005
20	4	12	5	elu	10	0.513 ± 0.005	0.00383 ± 0.00006
20	4	12	5	gelu	10	0.448 ± 0.007	0.0035 ± 0.0001
20	4	12	9	elu	10	0.526 ± 0.004	0.0041 ± 0.0001
20	4	12	9	gelu	10	0.454 ± 0.005	0.00366 ± 0.00005
20	4	12	11	elu	10	0.528 ± 0.003	0.0043 ± 0.0002
20	4	12	11	gelu	10	0.457 ± 0.007	0.00379 ± 0.00006
20	12	4	5	elu	10	0.625 ± 0.003	0.0065 ± 0.0001
20	12	4	5	gelu	10	0.589 ± 0.005	0.0068 ± 0.0002
20	12	4	9	elu	10	0.649 ± 0.004	0.0075 ± 0.0003
20	12	4	9	gelu	10	0.598 ± 0.007	0.0075 ± 0.0003
20	12	4	11	elu	10	0.650 ± 0.003	0.0081 ± 0.0001
20	12	4	11	gelu	10	0.595 ± 0.007	0.0079 ± 0.0002
20	12	12	5	elu	10	0.561 ± 0.005	0.027 ± 0.002
20	12	12	9	elu	10	0.557 ± 0.005	0.0275 ± 0.0007
20	12	12	11	elu	10	0.542 ± 0.003	0.027 ± 0.001

Table B.4: Result of PGD attack on the ensembles of BP-models: the number of models successfully attacked for each set of parameters, along with the average test accuracy and robustness within the group.

K	w	n	w_{pool}	σ	# models	Accuracies	$\mathcal{R}$
10	4	4	5	elu	10	0.630 ± 0.004	0.0073 ± 0.0002
10	4	4	5	gelu	10	0.589 ± 0.003	0.0063 ± 0.0001
10	4	4	9	elu	9	0.639 ± 0.003	0.0077 ± 0.0001
10	4	4	9	gelu	10	0.589 ± 0.003	0.0069 ± 0.0001
10	4	4	11	elu	9	0.629 ± 0.002	0.0081 ± 0.0001
10	4	4	11	gelu	9	0.581 ± 0.004	0.0072 ± 0.0001
10	4	12	5	elu	10	0.494 ± 0.005	0.00412 ± 0.00009
10	4	12	5	gelu	10	0.41 ± 0.01	0.00368 ± 0.00009
10	4	12	9	elu	10	0.511 ± 0.004	0.0048 ± 0.0002
10	4	12	9	gelu	10	0.421 ± 0.007	0.0040 ± 0.0002
10	4	12	11	elu	10	0.507 ± 0.004	0.0051 ± 0.0003
10	4	12	11	gelu	10	0.423 ± 0.004	0.00430 ± 0.00009
10	12	4	5	elu	10	0.614 ± 0.004	0.0080 ± 0.0001
10	12	4	5	gelu	10	0.571 ± 0.005	0.0078 ± 0.0001
10	12	4	9	elu	10	0.627 ± 0.006	0.0091 ± 0.0004
10	12	4	9	gelu	10	0.575 ± 0.008	0.0083 ± 0.0002
10	12	4	11	elu	10	0.621 ± 0.005	0.0096 ± 0.0003
10	12	4	11	gelu	9	0.567 ± 0.007	0.0085 ± 0.0002
10	12	12	5	elu	8	0.561 ± 0.005	0.0270 ± 0.0008
10	12	12	9	elu	10	0.548 ± 0.005	0.025 ± 0.003
10	12	12	11	elu	10	0.527 ± 0.006	0.024 ± 0.003
20	4	4	5	elu	10	0.646 ± 0.004	0.00572 ± 0.00008
20	4	4	5	gelu	10	0.614 ± 0.003	0.00537 ± 0.00006
20	4	4	9	elu	10	0.669 ± 0.002	0.00638 ± 0.00006
20	4	4	9	gelu	10	0.627 ± 0.004	0.0059 ± 0.0001
20	4	4	11	elu	10	0.668 ± 0.003	0.0067 ± 0.0001
20	4	4	11	gelu	10	0.626 ± 0.003	0.00621 ± 0.00004
20	4	12	5	elu	10	0.513 ± 0.005	0.00419 ± 0.00005
20	4	12	5	gelu	10	0.448 ± 0.007	0.0040 ± 0.0001
20	4	12	9	elu	10	0.526 ± 0.004	0.00431 ± 0.00009
20	4	12	9	gelu	10	0.454 ± 0.005	0.00390 ± 0.00003
20	4	12	11	elu	10	0.528 ± 0.003	0.0045 ± 0.0002
20	4	12	11	gelu	10	0.457 ± 0.007	0.00401 ± 0.00006
20	12	4	5	elu	10	0.625 ± 0.003	0.0067 ± 0.0001
20	12	4	5	gelu	10	0.589 ± 0.005	0.0069 ± 0.0002
20	12	4	9	elu	10	0.649 ± 0.004	0.0076 ± 0.0002
20	12	4	9	gelu	10	0.598 ± 0.007	0.0075 ± 0.0002
20	12	4	11	elu	10	0.650 ± 0.003	0.0081 ± 0.0001
20	12	4	11	gelu	10	0.595 ± 0.007	0.0078 ± 0.0001
20	12	12	5	elu	6	0.561 ± 0.005	0.026 ± 0.002
20	12	12	9	elu	10	0.557 ± 0.005	0.0257 ± 0.0006
20	12	12	11	elu	10	0.542 ± 0.003	0.025 ± 0.001

Bibliography

[1] Y. LeCun, Y. Bengio, and G. Hinton, "Deep Learning," *Nature*, vol. 521, pp. 436–44, May 2015.

[2] A. Madry, A. Makelov, L. Schmidt, D. Tsipras, and A. Vladu, "Towards Deep Learning Models Resistant to Adversarial Attacks," Sept. 2019. Comment: ICLR'18.

[3] S. G. Finlayson, H. W. Chung, I. S. Kohane, and A. L. Beam, "Adversarial Attacks Against Medical Deep Learning Systems," Feb. 2019.

[4] N. Arya, S. Saha, A. Mathur, and S. Saha, "Improving the robustness and stability of a machine learning model for breast cancer prognosis through the use of multi-modal classifiers," *Scientific Reports*, vol. 13, p. 4079, Mar. 2023.

[5] A. Ferdowsi, U. Challita, W. Saad, and N. B. Mandayam, "Robust Deep Reinforcement Learning for Security and Safety in Autonomous Vehicle Systems," in *2018 21st International Conference on Intelligent Transportation Systems (ITSC)*, pp. 307–312, Nov. 2018.

[6] B. Fung, "The technology behind the Tesla crash, explained," *The Washington Post*, July 2016.

[7] U. Orji, C. Ugwuishiwu, J. Nguemaleu, and Ugwuanyi, "Machine Learning Models for Predicting Bank Loan Eligibility," June 2022.

[8] C. Dwork, M. Hardt, T. Pitassi, O. Reingold, and R. Zemel, "Fairness through awareness," in *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*, ITCS '12, (New York, NY, USA), pp. 214–226, Association for Computing Machinery, Jan. 2012.

[9] C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. Goodfellow, and R. Fergus, "Intriguing properties of neural networks," Feb. 2014.

[10] J. Hoffman, D. A. Roberts, and S. Yaida, "Robust Learning with Jacobian Regularization," Aug. 2019. Comment: 21 pages, 10 figures.

[11] F. Rosenblatt, "The perceptron: A probabilistic model for information storage and organization in the brain," *Psychological Review*, vol. 65, pp. 386–408, 1958.

[12] L. G. Lindsay, *Models of the Mind: How Physics, Engineering and Mathematics Have Shaped Our Understanding of the Brain.* Bloomsbury Publishing (UK), Mar. 2021.

[13] J. Schmidhuber, "Deep learning in neural networks: An overview," *Neural Networks*, vol. 61, pp. 85–117, Jan. 2015.

[14] D. Krotov and J. J. Hopfield, "Dense Associative Memory for Pattern Recognition," Sept. 2016. Comment: Accepted for publication at NIPS 2016.

[15] J. J. Hopfield, "Neural networks and physical systems with emergent collective computational abilities.," *Proceedings of the National Academy of Sciences of the United States of America*, vol. 79, pp. 2554–2558, Apr. 1982.

[16] S. G. Brush, "History of the Lenz-Ising Model," *Reviews of Modern Physics*, vol. 39, pp. 883–893, Oct. 1967.

[17] Z. Uykan, "On the Working Principle of the Hopfield Neural Networks and its Equivalence to the GADIA in Optimization," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 31, pp. 3294–3304, Sept. 2020.

[18] D. Krotov and J. Hopfield, "Unsupervised Learning by Competing Hidden Units," *Proceedings of the National Academy of Sciences*, vol. 116, pp. 7723–7731, Apr. 2019.

[19] L. Grinberg, J. Hopfield, and D. Krotov, "Local Unsupervised Learning for Image Analysis," Aug. 2019.

[20] D. Patel and R. Kozma, "Unsupervised Features Extracted using Winner-Take-All Mechanism Lead to Robust Image Classification," in *2020 International Joint Conference on Neural Networks (IJCNN)*, pp. 1–7, July 2020.

[21] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[22] P. Mehta, M. Bukov, C.-H. Wang, A. G. R. Day, C. Richardson, C. K. Fisher, and D. J. Schwab, "A high-bias, low-variance introduction to Machine Learning for physicists," *Physics Reports*, vol. 810, pp. 1–124, May 2019. Comment: Notebooks have been updated. 122 pages, 78 figures, 20 Python notebooks.

[23] G. Cybenko, "Approximation by superpositions of a sigmoidal function," *Mathematics of Control, Signals and Systems*, vol. 2, pp. 303–314, Dec. 1989.

[24] K. Hornik, M. Stinchcombe, and H. White, "Multilayer feedforward networks are universal approximators," *Neural Networks*, vol. 2, pp. 359–366, Jan. 1989.

[25] M. Leshno, V. Y. Lin, A. Pinkus, and S. Schocken, "Multilayer feedforward networks with a nonpolynomial activation function can approximate any function," *Neural Networks*, vol. 6, no. 6, pp. 861–867, 1993.

[26] S. Sonoda and N. Murata, "Neural network with unbounded activation functions is universal approximator," *Applied and Computational Harmonic Analysis*, vol. 43, pp. 233–268, Sept. 2017.

[27] M. J. Colbrook, V. Antun, and A. C. Hansen, "The difficulty of computing stable and accurate neural networks: On the barriers of deep learning and Smale's 18th problem," *Proceedings of the National Academy of Sciences*, vol. 119, p. e2107151119, Mar. 2022.

[28] Izaak Neutelings, "Neural networks," 2022. https://tikz.net/neural_networks. This work is licensed under the CC BY-SA 4.0 license. To view a copy of this license, visit https://creativecommons.org/licenses/by-sa/4.0/.

[29] X. Li, Y. Grandvalet, and F. Davoine, "Explicit Inductive Bias for Transfer Learning with Convolutional Networks," in *Proceedings of the 35th International Conference on Machine Learning*, pp. 2825–2834, PMLR, July 2018.

[30] P. E. Utgoff, *Machine Learning of Inductive Bias*. Springer Science & Business Media, Dec. 2012.

[31] Janosh Riebesel, “Convolution operator,” 2022. https://tikz.net/conv2d. This work is licensed under the CC BY-SA 4.0 license. To view a copy of this license, visit https://creativecommons.org/licenses/by-sa/4.0/.

[32] I. Sobel and G. Feldman, “A 3×3 isotropic gradient operator for image processing,” *Pattern Classification and Scene Analysis*, pp. 271–272, Jan. 1973.

[33] X. Glorot, A. Bordes, and Y. Bengio, “Deep Sparse Rectifier Neural Networks,” in *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, pp. 315–323, JMLR Workshop and Conference Proceedings, June 2011.

[34] K. He, X. Zhang, S. Ren, and J. Sun, “Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification,” Feb. 2015.

[35] D.-A. Clevert, T. Unterthiner, and S. Hochreiter, “Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs),” Feb. 2016. Comment: Published as a conference paper at ICLR 2016.

[36] D. Hendrycks and K. Gimpel, “Gaussian Error Linear Units (GELUs),” July 2020. Comment: Trimmed version of 2016 draft; add exact formula.

[37] R. Pascanu, T. Mikolov, and Y. Bengio, “On the difficulty of training Recurrent Neural Networks,” Feb. 2013. Comment: Improved description of the exploding gradient problem and description and analysis of the vanishing gradient problem.

[38] X. Glorot and Y. Bengio, “Understanding the difficulty of training deep feedforward neural networks,” 2010.

[39] H. Li, Z. Xu, G. Taylor, C. Studer, and T. Goldstein, “Visualizing the Loss Landscape of Neural Nets,” Nov. 2018. Comment: NIPS 2018 (extended version, 10.5 pages), code is available at https://github.com/tomgoldstein/loss-landscape.

[40] D. P. Kingma and J. Ba, “Adam: A Method for Stochastic Optimization,” Jan. 2017. Comment: Published as a conference paper at the 3rd International Conference for Learning Representations, San Diego, 2015.

[41] I. Loshchilov and F. Hutter, “Decoupled Weight Decay Regularization,” Jan. 2019. Comment: Published as a conference paper at ICLR 2019.

[42] A. G. Baydin, B. A. Pearlmutter, A. A. Radul, and J. M. Siskind, “Automatic differentiation in machine learning: A survey,” Feb. 2018. Comment: 43 pages, 5 figures.

[43] “Hebb, d. o. the organization of behavior: A neuropsychological theory. new york: John wiley and sons, inc., 1949. 335 p. $4.00,” *Science Education*, vol. 34, no. 5, pp. 336–337, 1950. https://onlinelibrary.wiley.com/doi/abs/10.1002/sce.37303405110.

[44] D. Sterratt, B. Graham, A. Gillies, and D. Willshaw, *Principles of Computational Modelling in Neuroscience*. Cambridge University Press, 2011.

[45] B. W. Knight, “Dynamics of encoding in a population of neurons,” *The Journal of General Physiology*, vol. 59, pp. 734–766, June 1972.

[46] E. Oja, “Simplified neuron model as a principal component analyzer,” *Journal of Mathematical Biology*, vol. 15, pp. 267–273, Nov. 1982.

[47] T. D. Sanger, “Optimal unsupervised learning in a single-layer linear feedforward neural network,” *Neural Networks*, vol. 2, pp. 459–473, Jan. 1989.

[48] E. L. Bienenstock, L. N. Cooper, and P. W. Munro, “Theory for the development of neuron selectivity: Orientation specificity and binocular interaction in visual cortex,” *Journal of Neuroscience*, vol. 2, pp. 32–48, Jan. 1982.

[49] R. J. McEliece, E. C. Posner, E. R. Rodemich, and S. S. Venkatesh, “The capacity of the Hopfield associative memory,” *IEEE Transactions on Information Theory*, vol. 33, pp. 461–482, July 1987.

[50] “MNIST handwritten digit database, Yann LeCun, Corinna Cortes and Chris Burges.” http://yann.lecun.com/exdb/mnist/.

[51] “CIFAR-10 and CIFAR-100 datasets.” https://www.cs.toronto.edu/~kriz/cifar.html.

[52] H. Zhang, Y. Yu, J. Jiao, E. P. Xing, L. E. Ghaoui, and M. I. Jordan, “Theoretically Principled Trade-off between Robustness and Accuracy,”

[53] D. Tsipras, S. Santurkar, L. Engstrom, A. Turner, and A. Madry, “Robustness May Be at Odds with Accuracy,” Sept. 2019. Comment: ICLR’19.

[54] V. Antun, F. Renna, C. Poon, B. Adcock, and A. C. Hansen, “On instabilities of deep learning in image reconstruction and the potential costs of AI,” *Proceedings of the National Academy of Sciences*, vol. 117, pp. 30088–30095, Dec. 2020.