ZURICH UNIVERSITY OF APPLIED SCIENCES

MASTER THESIS

# Leveraging Neuroscience for Deep Learning Based Object Recognition

*Author:*
Claude LEHMANN

*Supervisor:*
Prof. Dr. Thilo STADELMANN
Prof. Dr. Christoph VON DER MALSBURG

*A thesis submitted in fulfillment of the requirements for the degree of Master of Science*

*in the*

Computer Vision, Perception and Cognition Group
Centre for Artificial Intelligence

11th July 2022

Zurich University
of Applied Sciences

zh
aw
School of
Engineering

# DECLARATION OF ORIGINALITY

## Master's Thesis at the School of Engineering

By submitting this Master's thesis, the undersigned student confirms that this thesis is his/her own work and was written without the help of a third party.

The student declares that all sources in the text (including Internet pages) and appendices have been correctly disclosed. This means that there has been no plagiarism, i.e. no sections of the Master's thesis have been partially or wholly taken from other texts and represented as the student's own work or included without being correctly referenced.

Any misconduct will be dealt with according to paragraphs 39 and 40 of the General Academic Regulations for Bachelor's and Master's Degree courses at the Zurich University of Applied Sciences (Rahmenprüfungsordnung ZHAW (RPO)) and subject to the provisions for disciplinary action stipulated in the University regulations.

| | |
|---|---|
| City, Date: | Signature: |
| Henggart, 11.07.2022 | [illegible] |

The original signed and dated document (no copies) must be included after the title sheet in all ZHAW versions of the Master's thesis submitted.

ZURICH UNIVERSITY OF APPLIED SCIENCES

# *Abstract*

School of Engineering
Centre for Artificial Intelligence

Master of Science

**Leveraging Neuroscience for Deep Learning Based Object Recognition**

by Claude LEHMANN

Whether it is the tv show we watch on a streaming platform after a long day or the *"people who bought X also bought..."* recommendations in a web shop, neural networks have a major impact on many of the decisions in our daily lives. These algorithms are hungry for the immense amount of data collected every day, but is the training of increasingly large networks the only path to success?

When comparing the training efficiency of a human to large deep learning models, we observe a gigantic discrepancy in sample efficiency between the two. While it takes children a handful of examples to recognize animals, deep neural networks typically need millions of examples to learn which pixels are indicative of a cat.

At its core, deep learning is about finding the best representations upon which a multitude of tasks are enabled. And while we cannot ignore the vast knowledge passed along through culture and genetics, we still believe that there must exist more efficient ways to train neural networks — and specifically, learn representations.

We explore ideas from neuroscience and deep learning to learn a novel representation with increased robustness. Machine learning has based many of its successes on the biological role model that is our brain and we believe both fields could benefit from a closer relationship. After all, the goal is to reproduce human intelligence, and chances are that it is best achieved by understanding the human mind.

In this thesis, we present the laterally connected layer (LCL), a novel neural network layer architecture that uses lateral intra-layer connections that are formed using the Hebbian rule during the forward pass. The lateral connections use the notion of self-organization, meaning they adjust themselves dependent on the input.

We show experimentally that a robustness increase can be achieved for object recognition using the popular MNIST dataset. We show that for a small accuracy reduction of 1% (from 98.28% to 97.30%), the performance on corrupted images can be increased by as much as 21%, specifically for noisy types of corruptions that are either contained locally or independent of neighboring pixels.

ZÜRCHER HOCHSCHULE FÜR ANGEWANDTE WISSENSCHAFTEN

# *Zusammenfassung*

School of Engineering
Centre for Artificial Intelligence

Master of Science

**Leveraging Neuroscience for Deep Learning Based Object Recognition**

von Claude LEHMANN

Unabhängig davon, ob wir nach einem langen Tag eine Fernsehsendung auf einer Streaming Plattform anschauen, oder *"Kunden die X kaufen, kaufen auch ..."* Vorschläge in einem Webshop sehen, so sind es neuronale Netzwerke, die alltägliche Entscheidungen massiv beeinflussen. Jedoch stellt sich die Frage, ob das Trainieren von immer grösser werdenden Netzwerken der einzige Weg zum Ziel sein kann.

Wenn wir die Lerneffizienz eines Menschen mit derer von grossen Deep Learning Modellen vergleichen, so sehen wir eine grosse Diskrepanz in der Stichprobeneffizienz der beiden. Während Kinder eine Handvoll Beispiele benötigen, um Tier zu erkennen, benötigen Deep Learning Modelle in der Regel Millionen von Beispielen, um zu lernen, welche Pixel relevant sind um eine Katze zu erkennen.

Bei Deep Learning geht es hauptsächlich darum, eine ideale Repräsentation zu lernen, welche ermöglicht, eine Vielzahl von weiteren Aufgaben zu lösen. Und obwohl wir das enorme Wissen, welches über Kultur und Genetik weitergegeben wird, nicht ignorieren können, so sind wir überzeugt, dass es effizientere Wege geben muss, neuronale Netze zu trainieren und Repräsentationen zu lernen.

Dazu erforschen wir Ideen aus den Neurowissenschaften und Deep Learning, um neuartige Repräsentationen mit verbesserter Robustheit zu lernen. Viele der Erfolge im maschinellen Lernen basieren auf dem biologischen Vorbild unseres Gehirns und wir sind überzeugt, dass beide Fachbereiche von einer Zusammenarbeit profitieren könnten. Das langfristige Ziel besteht schliesslich darin, die menschliche Intelligenz nachzuahmen, und es ist sehr wahrscheinlich, dass dies am besten durch das Verständnis des menschlichen Gehirns erreicht werden kann.

In dieser Arbeit stellen wir die lateral vernetzte Schicht (LCL) vor, eine neuartige Architektur, welche laterale Verbindungen innerhalb der Schicht verwendet. Die Verbindungen werden während dem Vorwärtsdurchgang unter Verwendung der Hebb́schen Regel gebildet. Die lateralen Verbindungen nutzen das Konzept der Selbstorganisation, das heisst, sie passen sich selbstständig über den Input an.

In unseren Experimenten zeigen wir, dass eine Robustheitssteigerung auf dem bekannten MNIST Datensatz erzielt werden kann. Wir zeigen, dass beim in Kauf nehmen einer kleinen Reduktion der Genauigkeit von 1% (von 98.28% auf 97.30%) die Genauigkeit bei beschädigten Bildern um bis zu 21% gesteigert werden kann. Insbesondere bei Bildfehlern basierend auf Rauschen, welches entweder lokal angewendet wurde oder unabhängig der benachbarten Pixeln ist.

# *Acknowledgements*

I'd like to take this opportunity to thank everyone involved in shaping my scientific career and supporting me through all the ups and downs. I thank . . .

Professor Thilo Stadelmann, my supervisor, who inspired me to follow my interests and pursue a career in academia. Your guidance helped me focus on bottlenecks and work at them with scientific rigor. Professor Christoph von der Malsburg, my co-supervisor, for his endless patience in explaining his thoughts and ideas, for providing insight in the fascinating world of neuroscience and for providing the general topic for this thesis. Professor Kurt Stockinger for all his feedback on writing scientific publications — such as this one. Dr. Pavel Sulimov for all the discussions on GPU optimization, the evaluation of dry erasers and Russian literature. Dr. Jan Deriu for the discussions on Hebbian learning and applying the laterally connected layer in more aggressive ways. Amirian Mohammadreza for his insights around convolutional neural networks and the analysis thereof. Kyle Tugwell for his help in setting up Unity and Blender to generate a synthetic image dataset. Patrick Lehmann, my brother, for teaching me how to code back when my most sophisticated programming skill was Microsoft Excel's VBA. My friends and my family, for giving me the space and time to dive deeply into this thesis' topics.

I thank each and everyone of you!

# Contents

# List of Figures

# List of Tables

*This thesis is dedicated to the memory of my grandfather Max Jeger. While we were not fortunate enough to meet in person, we are connected in our love for science nonetheless.*

# 1 Introduction

## 1.1 Motivation

*"We can only see a short distance ahead,*
*but we can see plenty there that needs to be done."*
– Alan M. Turing, *Computing Machinery and Intelligence* [1]

We are now living in a golden age of information. Many of our teachers used to torment us with the phrase "you're not always going to have a calculator at hand", but in current times, the vast knowledge of the human race through the internet is at our fingertips wherever we go. While my parents needed to go to the library and were limited by the availability of books, the knowledge bottleneck today is the speed at which we can feed information through the senses into our brain.

This vast amount of information is not just available to us humans, however. Many processes are automated through this data, supporting our daily choices. Which movie are we watching tonight? What products and offers am I getting recommended? Will the bank give me a mortgage? Will my driving behavior have an impact on my car insurance? These decisions are largely driven by machine learning systems that are trained on the plethora of data available — be it your browsing history, the current weather forecast, or the health data from your smart watch.

Since all these systems are heavily dependent on extracting as much information from the flood of data as possible, it becomes increasingly important to pore over data efficiently. In an age where our civilization must tackle challenges such as climate change, we should use our resources sensibly and not add more oil to the fire. On the other hand, the rules of physics are severely limiting processors from becoming even smaller, preventing us from continuing along a linear trajectory of Moore's law — at least in its original conception [2]. Thankfully, the emergence of very specialized hardware has allowed machine learning (and in particular deep learning) approaches to tackle immensely large amounts of data. However, we cannot solve the problem of scalability solely through faster and more capable hardware alone. After all, many state of the art models already consume significant compute and power resources [3] with training times measured in thousands of days [4, 5].

This thesis aims to take a step back and rethink current trends — is scaling up the only way forward, or should we instead explore different paths? Neuroscience inspired the first models of artificial neural networks as our brains are hyper efficient *real* neural networks, so what are further lessons to be learned? Deep learning today still uses the concept of neurons and synapse-like connections, though diverging from a biologically feasible model in many ways. Bridging the gap between artificial intelligence and neuroscience thus builds the foundation of this thesis.

## 1.2 Problem Statement

Modern machine learning, and in particular deep neural networks (DNN), have shown proficiency in a large variety of tasks. From DeepMind's AlphaGo [6] defeating the best human Go players, over OpenAI's language models GPT-2 and GPT-3 [5, 7] writing articles about newly discovered unicorn races [8] to Google's Imagen model [9] creating impressively detailed images just from a text caption [10], it could seem that the capability of deep learning models is unlimited and only increasing.

More critical voices were quickly able to find flaws in those systems, such as asking for an image of a horse riding an astronaut where the model showed images of the more familiar constellation of an astronaut riding a horse [11]. Similarly, if you ask GPT-3 for a step-by-step answer oh how long it takes for a dead cow to be alive again, it will happily come up with an estimate of about nine months [12]. It is possible to come up with additional counter examples, though they share the same characteristic: A situation that is seldom (if ever) seen in the training data. Training datasets try to reflect our world and contain an implicit bias — common sense. While a corpus of text has a non-zero probability of containing fiction, the majority of occurrences of objects riding horses will be in that particular order. Furthermore, there is an almost infinite space of objects pairs to fit the `A rides B` relationship, which even immensely large datasets are not able to cover realistically.

These counter examples to the apparent near-mastery of the trained tasks does not necessarily diminish the models' accomplishments — after all, *to err is human*, and aren't we trying to replicate human behavior? — but they let us see through the cracks. All these models were trained on immense amounts of data, in the case of GPT-3 about 45 terabytes (TB) of text. To put that into context, if printed on A4 sheets of paper we could stack them to reach the international space station (ISS) 21 times or cover one sixth of Switzerland in paper. At 250 words per minute, it would take approximately 5'000 human lifetimes solely spent reading to go through this dataset just once. Clearly, we humans cannot compete with this amount of training data, but we definitively still comprehend language better than them.

The efficiency at which we humans learn from a *much* smaller dataset during infancy points towards significantly more efficient learning methods that are not mimicked by current deep learning architectures. From this we derive the flaws of deep learning, which guide us towards the research question of this thesis:

- **Sample efficiency.** Deep learning systems must reduce their dependence on seeing samples in every configuration. To recognize a dog, a handful of samples should suffice, rather than exposing the system to millions of dog images across all breeds, sizes, and environments. Accomplishing this goal would allow such systems to learn faster and require significantly smaller datasets.

- **Robustness.** Deep learning systems should be able to recognize patterns even for disturbed input signals. A teacup remains a teacup even if it is viewed from a different angle or is partially obstructed. Current systems overcome this issue through scaling up the dataset and reducing the chance for situation that are not encountered during training. Even so, the probability of encountering out of distribution examples is only reduced and not averted completely[1].

- **Generalization towards common sense.** In the current state, deep learning systems are trained towards solving well defined tasks (e.g. playing Go, filling

[1]Data security topics such as membership inference or training data poisoning even rely on this.

in blanks in sentences or learning categories of images); however, they rely on human intervention to set up the task to be learned. Deep learning systems should train a meta-task of common sense, to enable better decisions.

The goal of this thesis is to present a novel neural network architecture that addresses these flaws and shows both theoretically and experimentally, that it generates a fundamentally different representation. While solving all points raised above is beyond the scope of a master thesis, we aim to answer one question:

*How can we learn a more robust representation?*

## 1.3 Contributions

This thesis makes the following contributions:

- We give an overview of concepts from machine learning and neuroscience to bridge a gap between both areas. This contribution is presented in Chapter 2.
- We propose a novel neural network architecture, that improves the robustness on computer vision tasks by introducing the notion of lateral intra-layer connections. This contribution is presented in Chapter 3.
- We introduce a novel training paradigm for learning statistical correlations from input signals using Hebbian learning. This unsupervised process takes part in the forward pass rather than backpropagating an error through the network. This contribution is also presented in Chapter 3.
- We conduct experiments to show the robustness improvements on the task of recognizing handwritten digits. This contributions is presented in Chapter 4.

## 1.4 Organization of this Thesis

The remainder of this thesis is structured as follows:

- Chapter 2 gives an overview of relevant literature and concepts, to bring both fields of machine learning and neuroscience together. In addition, this chapter allows to discuss the benefits of individual methods (especially Hopfield networks) and strengthens the foundation for the following chapters.
- Chapter 3 introduces the novel neural network layer architecture of the laterally connected layer, which incorporates learning mechanisms that more closely resemble how humans learn and form the connections in the brain. It also contains reasoning for many of the design choices on the theoretical level and gives room to discuss the design goal of the individual characteristics.
- Chapter 4 gives room to review the hyperparameters of our novel layer. A multitude of experiments are performed to evaluate the performance of a traditional convolutional neural network against the same network with our layer.
- Chapter 5 summarizes the results of Chapter 4, discusses their significance and shows which avenues are promising for future work.

- The Appendix contains superfluous information about the codebase used for implementing our experiments and the lateral connected layer. It also covers our empirical studies to understand the inner workings of the novel architecture and features visualizations of the experimental results of Chapter 4.

# 2

# Fundamentals

In this chapter, we introduce a variety of fundamental concepts, which will be relevant for the following chapters of this thesis. The goal is to cover ideas from neuroscience and machine learning, to build a common foundation.

## 2.1 Deep Learning

In its essence, deep learning is a statistical modelling technique that uses a large number of labelled training samples (the dataset) to solve pattern recognition tasks using neural networks with many layers. These neural networks are trained on vast amounts of data, optimizing for a loss (or target) function using backpropagation [13] and converging towards an optimal set of connection weights.

### 2.1.1 Neural Networks

The basis for artificial neural networks (ANN) comes from biology, aiming to recreate the interaction of brain cells (neurons) with a mathematical model. Given that this network structure of neurons and synapses governs the complexity of our human minds, an artificial replica should be able to achieve similar successes — while in turn giving us the ability to further understand our own brain.

A first such model was the McCulloch and Pitts neuron model [14]. It takes binary values and calculates the sum of its input. The neuron outputs 1 if the sum reaches a threshold and 0 otherwise. Rosenblatt [15] later developed the Perceptron, allowing inputs to be any real number. Furthermore, the Perceptron applies weights to each input and calculates the linear combination of weights and inputs. Finally, the activation function (here the binary step function) is applied to the sum.

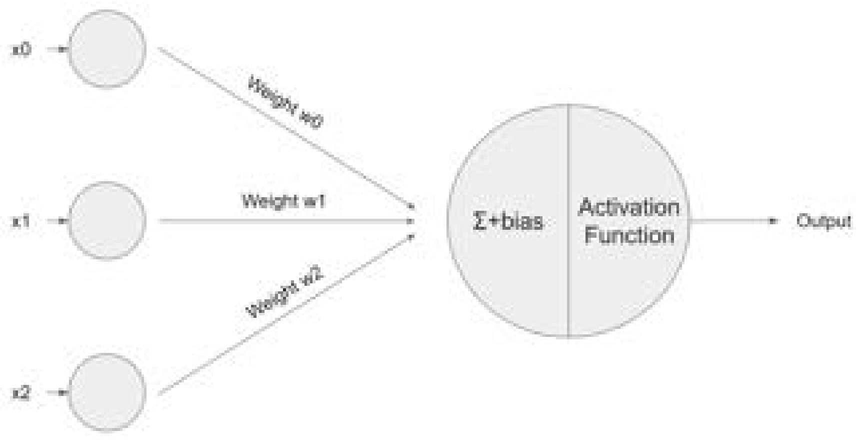

FIGURE 2.1: An example of Rosenblatt's Perceptron model [15].

More generally, an artificial neural network is a network of neurons that have synapse-like connections. Unlike in a graph of people and their relationships, neurons in an ANN are grouped into layers, forming a directed graph structure without cycles (recurrent neural networks (RNN) are an exception to this rule, as the output is fed back into the neuron as part of its input). The term multilayer perceptron (MLP) is often used to describe the general architecture of perceptron neurons arranged in layers. Any input travels successively through all layers in a fixed order with neurons from layer $l$ only receiving an input from the previous layer $l-1$. Every connection between two neurons has a distinct weight associated with it. Equation 2.1 shows how the output of a neuron $j$ is calculated. $w_i, j$ corresponds to the weight between neuron $i$ and neuron $j$, $x_i$ is the output value of neuron $i$. The bias term $b_j$ moves the threshold away from zero, allowing for arbitrary thresholds. The activation function $f$ is then applied to the sum to generate the final output.

$$a_j = f\left( (\sum_{i=0}^{n} w_i, jx_i) + b_j \right) \tag{2.1}$$

The main goal of the activation function is to both act as a threshold and allow the learned model to exhibit nonlinear properties. Early models generally used the sigmoid function $\sigma(x)$ shown in Equation 2.2, as it was well known from logistic regression and possesses a smooth derivative. Depending on the task and the model architecture, a variety of activation functions have shown to be effective[1].

$$\sigma(x) = \frac{1}{1+e^{-x}} \tag{2.2}$$

**Network Training**

ANNs are trained using algorithms from three groups: supervised, unsupervised (sometimes also referred to as self-supervised) and reinforcement learning. In supervised learning, tagged or labelled examples are presented to the model and its prediction is compared to the desired output, calculating the error difference, and using it as a signal to adapt the model's weights to better produce the desired output. In unsupervised learning, there are no labels available, so the models extract patterns in the data. Lastly, reinforcement learning describes a constant loop between an agent choosing actions based on an expected reward and an environment responding with how the change affected the environment and what reward was earned.

Back to supervised learning, the connection weights need to be adjusted such that for any given input signal the resulting output matches our expectations. The input $x$ is given to the network and fed through all the layers until an output $\hat{y}$ is calculated. Knowing what the output $y$ should look like, we can calculate the observed error $e(y, \hat{y}) = y - \hat{y}$. The error magnitude should ideally converge towards zero. The network weights are updated according to the gradient of the error, pointing in the direction of steepest ascent. By adjusting the weights towards the negative gradient, the iterative training process successively approaches a better state. If the network only consists of a single layer, this step is rather straight forward. However, many ANNs are built from a large number of hidden layers (i.e. layers that are between the input and output layers) that each directly influence the calculation of the next layer. Because of this strict dependency, the error gradient is propagated in

[1]While a detailed discussion on the different activation functions is out of scope for this thesis, we refer the interested reader to a recent review article [16].

reverse order starting from the output layer. This allows the weights to be updated in each layer relative to the impact of the layer on the overall output.

Loss functions further improve the process of backpropagation, by changing how large the error is in various configurations. Instead of only calculating the error by subtracting the output (or prediction) from the desired output, loss functions can further increase or decrease the error term. For example, if a network is expected to return the probability of any presented image showing a cat, it would output a probability between 0 and 1. When showing an image without a cat, seeing a prediction probability of 1 versus a probability of 0.5 leads to an error of $e(0,1) = 0 - 1 = -1$ and $e(0,0.5) = 0 - 0.5 = -0.5$. The first case is thus twice as bad as the second case. Using the squared function $f(x) = x^2$ for the error instead yields $e(0,1) = (0-1)^2 = 1$ and $e(0,0.5) = (0-0.5)^2 = 0.25$. Predicting that there is a cat with a 100% compared to 50% chance is now four times worse, which in turn punishes the network more strongly and updating the weights harsher in the first case. The mean squared error loss functions thus punishes outliers more harshly.

### 2.1.2 Convolutional Neural Networks

For visual tasks in particular, deep neural networks use convolutional layers in addition to fully connected layers. A neural network that employs convolutional layers is referred to as a convolutional neural network (CNN). Instead of connecting each input neuron with each output neuron (and assigning a weight to all connections), the convolutional layer uses an image kernel that slides across the two-dimensional input, calculating a local sum of the element-wise pixel product. Because the same kernel is used for the whole image, a much smaller number of weight parameters needs to be trained and managed. In addition, sliding the kernel across the image allows this layer to pick up identical patterns in different locations of an image, giving it translational invariance. The output of a convolutional layer is called a feature map. CNNs often train a large number of kernels in each of their convolutional layers, producing as many feature maps as there are filter kernels.

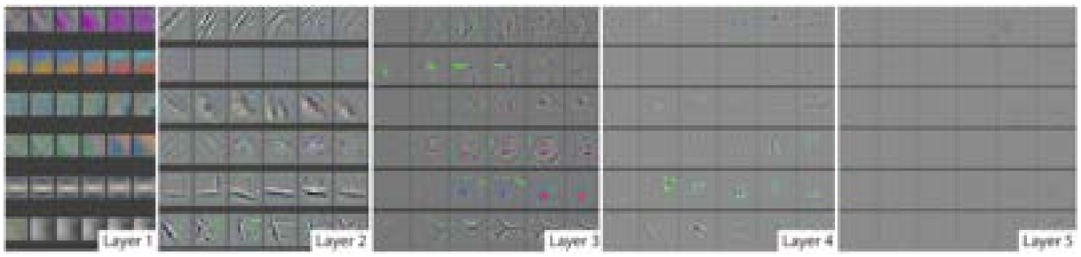

FIGURE 2.2: Various feature activations across the first five layers of a convolutional neural network with increasingly complex features [17].

Furthermore, Zeiler and Fergus showed that CNNs converge to kernels with structures of increasing detail, in relation to the position of the convolutional layer across the path through the network [17]. That means, the first layer learns to recognize very simple structures, such as edges and blobs. The second layer starts to combine previous patterns to form corners or curves. Finally, very complex patterns are formed that are starting to be recognizable by us as objects - for example a face or and eye. Figure 2.2 by the authors illustrates it for the first five layers of a CNN.

As with the perceptron model, the idea for CNNs stems from biology. Hubel and Wiesel [18–20] discovered, that neurons in the cortex of cats spiked when certain patterns were shown. They experimented with the receptive field of a cat's vision and encountered accidentally that the neurons would fire for edges of glass plates on the projector. This led to the discovery, that some neurons specialize to very distinctive patterns, such as edges in one particular orientation. Different neurons would only activate for complicated patterns, so the authors introduced the idea of simple and complex cells. This combination allows to build pattern detectors through a hierarchical combination of simple cells aggregating up to complex cells. Fukushima implemented these concepts with the Neocognitron [21], a model that uses convolutional and downsampling layers. The convolutional layers gave the ability to detect patterns even if they were rotated or shifted, while the downsampling layers increased the receptive field across the network. LeCun improved upon the Neocognitron by using backpropagation of errors for training [22].

### 2.1.3 Spiking Neural Networks

A variant that is closer to our biological neurons are spiking neural networks (SNN). Their main difference is the addition of time. Neurons only fire once their potential reaches a threshold value. Generated spikes interact with other neurons that are either excited or inhibited, adjusting their potential. When a triggered neuron's potential crosses the threshold, it also sends out a spike, which can result in a tidal wave of spikes. As with our biological neurons, the potential is lowered after a spike as not to constantly fire. Some models even go as far as to introduce a refractory period, where neurons are completely prohibited from firing following a spike.

Among the most used spiking neural networks are the (leaky) integrate-and-fire models [23] introduced over hundred years ago. Of course, back then these models were not run on a computer but were built as an electrical circuit, where a firing neuron would lose all its charge and reset to a default voltage. An interesting difference to other neural networks is that the curve shape of the spiking voltage (be it a circuit or a value on a computer) is not relevant, but instead the existence or absence of spikes. The resulting output of neuron spikes is referred to as a spike train.

## 2.2 Synaptic Plasticity

The term synaptic plasticity describes the ability of synapses to rewire and change how strong their connection is. The main mechanisms are described below:

### 2.2.1 Hebbian Plasticity

Hebb introduced the often quoted saying that "cells that fire together, wire together; cells that fire out of sync, lose their link" [24]. This is a succinct summary of Hebbian plasticity, describing a feedback loop dependent on the firing timings in neurons. Given that two neurons activate in short succession, their synapse grows stronger and increases the probability for those neurons to fire together in the future. This process is a positive reinforcing feedback loop that builds up momentum.

### 2.2.2 Homeostatic Plasticity

Opposing the nature of Hebbian plasticity's positive feedback loop, homeostatic plasticity aims to regulate Hebbian plasticity by scaling synaptic weights in accordance with the activity of the neuron and the overall network. The refractory period also belongs to homeostatic plasticity, as it limits how quickly a neuron can fire again.

## 2.3 Associative Memory

### 2.3.1 Hopfield Networks

Over 40 years ago, Hopfield introduced a recurrent neural network architecture [25], which is now known as a Hopfield network. This new model is inspired by biological organisms, where from a simple rule set complex behavior emerges (i.e. the flocking behavior of animals [26] or the defensive wing strokes of honey bees [27]).

Hopfield networks [25] are an associative memory system, where binary states are stored and retrieved. Hopfield networks have only a single layer, simultaneously acting as its own output layer. Every neuron is symmetrically connected to every other neuron in the layer. While the signals travel synchronously through a feed-forward network, updates in Hopfield networks can be performed asynchronously.

The binary state vector $V$ with values $v \in \{0, 1\}$ acts both as input and output of the network[2]. The connection strengths (or weights) between neurons $i$ and $j$ are stored as $w_{i,j}$. The weights $w_{i,j}$ are symmetric (i.e. $w_{i,j} = w_{j,i}$) and there exist no self-connections (they are represented as zero strength connections $w_{i,i} = 0$).

To store a set of $n$ states $V^s$ with $s \in 1, ..., n$, the weight matrix W is adjusted with the formula in Equation 2.3. States can be added and removed independently, to remove a state $V_s$ simply subtract the term $(2v_i^s - 1)(2v_j^s - 1)$ from the weights $w_{i,j}$.

$$w_{i,j} = \sum_s (2v_i^s - 1)(2v_j^s - 1) \tag{2.3}$$

Retrieving a state $V_s$ requires "any subpart of sufficient size" [25]. For example, a Hopfield network with 6 neurons and the stored patterns $V_1 = 101001$ and $V_2 = 001100$ should retrieve $V_2$ based on similarity given an initial state of $V_{initial} = 000000$.

Retrieval is done by multiplying the current state of the network with the weights and evaluating, whether each neuron lies above or below a threshold $t_i$. In Hopfield's original paper, the threshold is generally chosen to be zero. Equation 2.4 shows how the individual neuron values $v_i$ of neuron $i$ are updated. To avoid situations where updates are stuck in an endless loop, updates should be performed sequentially and in random order. Neurons, for which their state value is known to be correct, can be omitted from the update process.

$$v_i = \begin{cases} 1 \text{ if } \sum_{j, j \neq i} w_{i,j} \, v_j > t_i \\ 0 \text{ otherwise} \end{cases} \tag{2.4}$$

Unlike in the neural networks discussed previously, where the minimization of a loss function is achieved through gradient descent and back propagation, Hopfield networks define an algorithmic update rule. However, the goal of applying the update is rule is also the minimization of a similar function, the Hopfield energy $E$,

[2]In the literature, the state values are often changed to be $v \in \{-1, +1\}$ in order to visually simplify the equations. However, these variants are functionally identical to the equations shown below.

given in Equation 2.5. Every stored pattern represents a local minimum for the energy function and updating a state through the update rule reduces its energy, moving the current state closer towards a stored state. One can think of a memory foam mattress, where patterns are pressed into the foam. The update rule then moves a ball along downwards facing grooves until a (local) minimum is reached.

$$E = -\frac{1}{2}\sum_{i}\sum_{j,j\neq i} w_{i,j} v_i v_j \tag{2.5}$$

**Limitations of Hopfield Networks**

Hopfield estimated the storage capacity — the number of patterns that can be stored without errors — of his Hopfield networks to be $C = 0.15N$ [25], "before error in recall is severe". Later work has shown that $0.15N$ was an overestimate and its value is closer to $C = 0.138N$ [28]. Intuitively it is reasonable that each stored pattern should have a sufficient distance between other patterns, such that any initial state is closest to one particular stored pattern. The smaller these distances between patterns become, the harder it becomes to differentiate which pattern should be retrieved.

If however, too many patterns are stored, the retrieved states can take the form of so called spurious states. These are either the negation of a stored pattern or become a combination of multiple stored patterns. Spurious states can be caused by a lack of capacity, or when patterns are too similar. Research showed that the state retrieval is more successful if states are less correlated or ideally orthogonal [29].

From a biological standpoint, the symmetry of weights $w_{i,j}$ and $w_{j,i}$ is a mathematical trick to keep the numbers at bay. In practice, it is doubtful that neurons for example in our brain connect with the same symmetric intensity for every synaptic connection. Furthermore, there is no biological need to fully connect every neuron with every other neuron available. For example, receptors in the retina are only connected to other receptors in their local neighborhood, but not on a global scale.

### 2.3.2 Hopfield Network Extensions

Many extensions to the original Hopfield network have been published over the years, from the continuous Hopfield networks [30] over the reduction of spurious states [31] to an increase in storage capacity [32, 33]. Krotov and Hopfield adjusted the energy function from $F(x) = x^2$ to any polynomial function $F(x) = x^n$, increasing the storage capacity greatly [32]. For $n = 2$ they again calculate the known capacity of $C = 0.14N$, but they show that for larger $n$, the capacity grows "in a non-linear way" with $n$ in the exponent as the major contributing factor of the following Equation 2.6 for the storage capacity. $n$ is the exponent of the energy function $F(x) = x^n$ and $N$ represents the number of neurons in the Hopfield network.

$$C \approx \frac{1}{2(2n-3)!!}\frac{N^{n-1}}{ln(N)} \tag{2.6}$$

While the increase in storage capacity is formidable, intuitively using values for $n > 2$ forces the Hopfield network to connect groups of up to $n$ neurons that interact at once (instead of just two in the original Hopfield network). A mathematical demonstration of the $n$-th order interaction of neurons is shown in the Appendix A.1.3.

Demercigil et al. took adjusting the energy function a step further and introduced an exponential energy function $F(x) = e^x$ [33], by taking the limit of Krotov and

Hopfield's polynomial energy function [32]. Their paper goes on to rigorously proof that their choice of $e^x$ is sound and retains the properties of the Hopfield network, while increasing the storage capacity in proportion to $c^{\frac{N-1}{4}}$ [34]. The implications of taking $n$ to infinity are explored in the Appendix A.1.4.

### 2.3.3 Modern Hopfield Networks

Ramsauer et al. [34] extend the exponential energy function [33] to continuous state patterns and introduce a novel update rule. Furthermore, they show that their new update rule can be rewritten as the attention mechanism of the very successful Transformer architecture [35], to which the paper title "Hopfield Networks is All You Need" pays homage to. The authors claim to reach an exponential storage capacity of $C = 2^{d/2}$ and provide additional information and proofs in their Appendix.

### 2.3.4 Introduction of Hidden Neurons

Krotov and Hopfield recently discussed the problem of using larger order energy functions, where connections between neurons are no longer between pairs, but between triplets (for the cubic function, or even larger sets of neurons for higher order polynomials) [36]. Triplet connections are not feasible in a biological system at large, so they suggest the incorporation of hidden units (as in deep learning). These would allow to keep the improvements in storage capacity described in the previous sections but bring synapses back to a biologically sensible pair-level.

## 2.4 Metamers & Foveated Imaging

Freeman and Simoncelli [37] coined the term *visual metamer* for sensory input to the visual system, where a human observer is unable to distinguish between two different images of the same scene. These metamer images use the fact that the receptive field scales with eccentricity [38, 39]. In other words, we detect the highest level of detail where we focus our eyesight and the further away something is from our focal point, the less detail we perceive. If any changes are applied to an image, we are much less likely to perceive them in away from our focus areas.

With foveated imaging, this characteristic of the human visual system is exploited. Using this technique, the images feature a higher level of detail around a single or multiple focal points. The remainder of the image is kept at an progressively lower level of detail, potentially saving bandwidth or storage space in general. The name originates from the fovea in the center of our eye's retina.

Figure 2.3 [37] shows examples of visual metamers. (a) shows an approximation of the receptive field sizes, (b) is the original image of the scene, (c) illustrates a strongly foveated and (d) a less strongly foveated metamer of the same scene.

Psychophysical experiments have been conducted with those images [37], where participants were shown white noise and then two different metamers (for example (b) and (d)) with white noise in between. They were then asked if the two images were the same or different, allowing the authors to find a threshold at which it was not possible to differentiate between two different images anymore. This threshold encapsulates a many-to-one mapping, where all images with changes that are smaller than the threshold are mapped to the same intrinsic representation in our mind. If we translate this idea into a latent vector representation for images, there exists a volume around each point, for which all images are perceived identically.

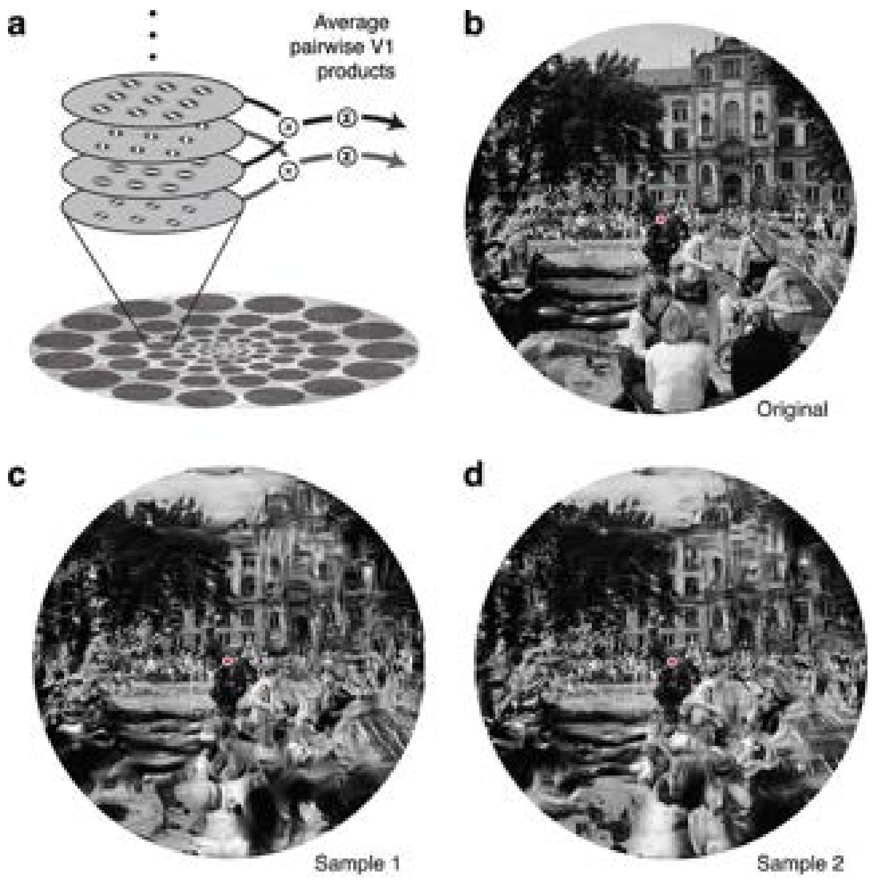

FIGURE 2.3: Example images of metamers as shown by Freeman and Simoncelli [37]. The original image (b) is unaltered, samples 1 (c) and 2 (d) are foveated to different degrees.

### 2.4.1 Object Recognition Through Hierarchical Building Blocks

Biederman [40] views object recognition as a process dealing with components. These components are defined through 36 primitive shapes (cylinders, spheres, wedges, etc.), called geons. Differentiating between the geon shapes is done primarily through five invariant or non-accidental properties (NAPs): curvature, collinearity (straight lines), symmetry, parallelism and cotermination (edges ending in the same vertex). In contrast to the NAPs are metric properties (MPs), such as length, aspect ratio, intersection angles or the degree of curvature. MPs can vary greatly between objects of the same class, meaning a strong object recognition method should be invariant to metric transformations. Biederman even hypothesizes, that if the arrangement of geons can be captured, the original object can be recognized. From this idea, the unique combination of geons would hierarchically identify any object in our world and be a prototypical unique identifier — a kind of object metamer.

### 2.4.2 Texture Synthesis: An Application of Metamers

Gatys et al. [41] presented a breakthrough in texture synthesis. They presented a new representation of texture images, which is calculated from pixel-wise statistical measurements (primarily correlations) between different layers and feature maps of a CNN (a VGG19 [42] to be precise), called the *gram matrix* $G^l$ at layer $l$. They have found that for texture images with very similar gram matrix values, images display the same kind of texture (see Figure 2.4 below for examples of this).

To synthesize a new texture of i.e., gravel, the gram matrix $G^l$ for the original image is calculated by running it through their CNN. A white noise image is then fed through the same network and its gram matrix $\hat{G}^l$ is also calculated. Through gradient descent, the necessary changes to the input image are back propagated through the network to iteratively converge $\hat{G}^l$ to become identical to $G^l$, at which point the white noise image has morphed into an image of similar looking gravel. Therefore I hypothesize that the gram matrix can be understood as a measurement for *texture metamers*, where similar textures are clustered together in the gram space. This idea is not necessarily new, over half a century ago Julesz showed that two different images of the same type of texture have very similar n-th order statistics [43].

For calculating the gram matrix $G^l$, the authors generate a $n \times n$ matrix, where $n$ is the number of feature maps of layer $l$. Each entry $G^l_{i,j}$ represents the sum of pixel-wise products, shown in Equation 2.7, across both feature maps $i$ and $j$. The $F^l_{i,k}$ value stands for the pixel of network layer $l$ in feature map $i$ at position $k$[3]. The calculated sum in $G^l_{i,j}$ is only bounded by the size of the feature maps in question. Individual values of the gram matrix are uninterpretable on their own and only gain meaning in relative comparisons. Even so, the authors showed that this measurement is meaningful enough to synthesize entirely new versions of the input texture.

$$G^l_{i,j} = \sum_k F^l_{i,k} * F^l_{j,k} \tag{2.7}$$

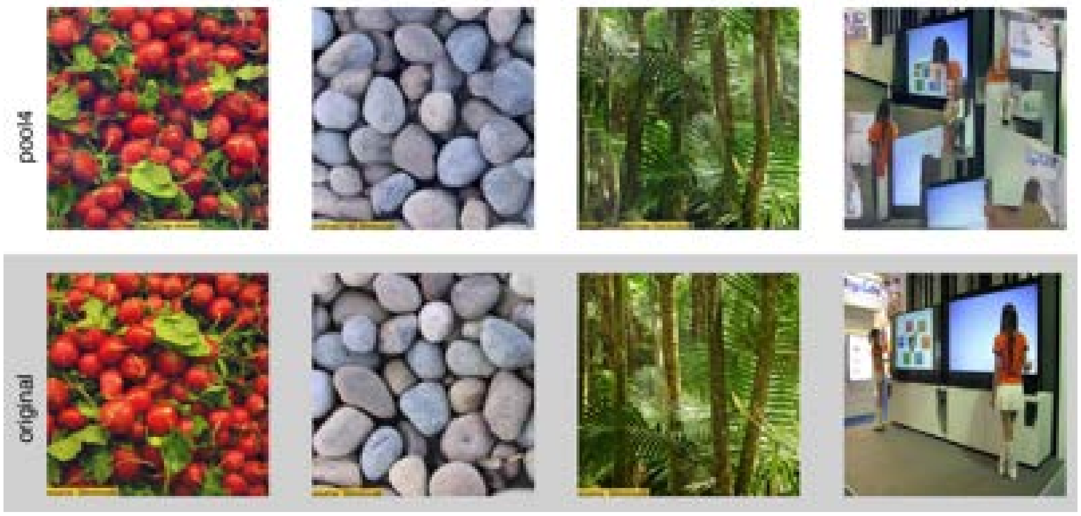

FIGURE 2.4: Original input image (bottom) and the synthesized texture (top), starting from a white noise image and using the gram matrix differences for gradient descent [41].

Figure 2.4 from the author's paper [41] show four example of this synthesis[4], where the bottom image is given to the network and the top image is synthesized from white noise. This technique works very well for textures without global structures, as shown in the first three columns of Figure 2.4. The right most column however illustrates a lack of global awareness, where large patches of the input image are textured together, losing a large degree hierarchical information in the process. The

[3]In practice, the feature maps are two-dimensional and $k$ iterates over both dimensions, encompassing all pixels of the feature map.

[4]We encourage the interested reader to take a look at the author's website with hundreds of additional astonishing examples here: `http://bethgelab.org/deeptextures/`

image shows a room with two screens on the wall and two persons standing next to them. The synthesized image loses the notion of a room and arranges patches randomly. Attentive readers can spot the same loss of global arrangement in the other three images as well, where the yellow label *'source: Simoncelli'* is moved across the image, partially obstructed and made unreadable in the first three columns.

The authors also conducted an experiment to investigate the outcome of exchanging the white noise initialization with structurally rich images, such as faces, leads to interesting outcomes, where the global structure of the to-be-synthesized image is kept and back propagating its gram matrix performs a mixture between texture synthesis and style transform. An example is shown in Figure 2.5 below.

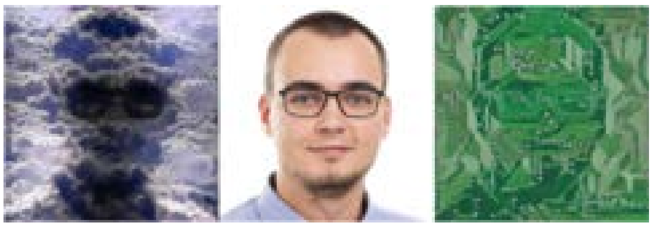

FIGURE 2.5: Performing texture synthesis on a profile picture of this thesis' author (center), using an image of a cloudy sky (left) and a motherboard (right) instead of white noise.

## 2.5 Related Work

### 2.5.1 Lateral Connections

Lateral connections have been used for spiking neural networks (SNN) where the biology shows exemplary how and where they should be implemented. It is less clear, how they should be employed in perceptron-based feed forward networks.

SNN have not seen the level of success compared to regular ANN or CNN architectures. An open problem remains how these models should be trained in a supervised setting, as current methods (e.g. Hebbian learning and spike-timing-dependent plasticity (STDP) [44]) are ultimately unsupervised. First attempts have been made to train a SNN in a supervised fashion through backpropagation [45].

Because of the similarity with our brain, lateral connections in SNN are a natural choice to connect neurons and propagate spikes locally, achieving good results for digit recognition [45, 46]. The authors of [46] also observe, that the lateral inhibition stabilizes the learning process such that no performance degradation can be seen for numbers of iterations where overfitting normally is observed. This finding gives credence to the hypothesis that lateral connections increase the robustness.

In feed-forward networks, lateral connections were proposed as part of a fully connected hidden layer [47]. Static connections are added from every $j$-th neuron to every $j+1$-th neuron to force a dependence on neurons in hidden layers. Their goal is to mitigate the herd effect [48], where neurons focus on reducing the most dominant error at the cost of less frequent errors, e.g., by focusing on the most prominent class in an imbalanced dataset. Added lateral connections were shown to be effective at making different hidden units target other sources of errors to mitigate.

### 2.5.2 Learning Algorithms

Backpropagation is the de-facto standard algorithm for training a neural network. Alternatives are the previously described Hebbian rule or genetic algorithms, where weights are evolved by trying multiple configurations and breeding the best performing ones to form a new generation of configurations [49].

Attempts have been made to introduce algorithms that propagate forward, for example in reinforcement learning [50]. The authors inject observed errors from the environment (e.g. steering too far from the road to be followed) into the input and calculate the errors on each layer alongside the signals. Weights are updated using the Hebbian learning rule while calculating the output of the network.

Another very recent publication uses Hebbian updates to train a neural network during the forward pass in a supervised setting [51]. Their approach reduces the necessary structural complexity of backward connections and removes the sequential nature of backpropagation-based learning where a backpropagated update step always follows a forward pass. In their forward signal propagation (FSP) learning algorithm, labels are passed through the same network as the input to generate targets at every level. The loss $L_i(h_i, t_i)$ is calculated between the output $h_i$ of the $i$-th layer and the target $t_i$ at layer $i$, without the need to wait for all subsequent layer's calculations first. Even though FSP does not outperform traditional backpropagation in their experiments, their results show a significant improvement in both time and memory, by rough factors of 2 and 4, respectively.

# 3 Method

In this chapter we present our prototype architecture, by first summarizing our inspiration, and second describing the processes for training and inferencing. The model outlined in this chapter is primarily based on implementing the ideas of Christoph von der Malsburg outlined in [52, 53]. While this thesis does not fully implement all aspects of their works, it should be seen as a steppingstone in exploring the possibilities of the proposed ideas in generating a different representation.

## 3.1 Inspiration

Many cells in the visual system of us humans either exhibit short or far-reaching connections, interacting with a local or global context of other cells. Classically, every neuron in an artificial neural network layer is only connected to neurons of the previous and following layers. Our goal is to replicate the behavior of the visual system, by breaking the typical feed-forward architecture of artificial neural networks and introducing intra-layer connections that connect neurons in local neighborhoods, that in turn correspond to local regions in the input images.

According to [52], excitatory connections make up the representation of memory in the brain. Synaptic plasticity governs the strength of these connections and controls it by up- and downscaling synapses in relation to their activity. As such, memories can be seen as subgraphs in the network of synapses, where existing memories build hierarchically upon the pre-existing structure of others. Network self-organization is the process to build this network structure of neurons that fits the statistical characteristics of the passed signals best. Each update is part of a larger feedback loop mechanism — Hebbian plasticity — where pairs of neurons that activated in close proximity (both geographically and temporally) increase their connection strength as well as their probability to activate in the future.

Implementing this mechanism of network self-organization in a deep learning model requires a new kind of neurons that exhibit the ability to form lateral connections. These connections should only allow to connect to neurons from the same layer and in a small local neighborhood. Generally speaking, it is not obvious what the local neighborhood represents for an arbitrary deep learning layer, so we will focus on the locality in a convolutional layer, where pixel values at similar coordinates represent signals from similar regions of the given input. The primary assumption is, that any input to our laterally connected layer (LCL) will be the feature map output of a convolutional layer (as part of a much larger CNN like the VGG19 architecture [42]) or describe similar characteristics (e.g. Gabor filters). Our eyes receive the input on the retina in a two-dimensional grid as well, so the limitation to work on feature maps is comparable to the limitation of our own visual system.

Our implementation takes inspiration from the proposed implementations [52], and in particular what they call rapidly switching connections: multi-cellular units.

Multi-cellular units (MCU) simply represent a set of identically repeated neurons. Each repeated neuron can have connections with the same set of different MCU neurons, but with different connection strengths. When activated, all repeated neurons in an MCU receive signals via their connections and only the strongest repeated neuron is chosen to be activated in a Winner-Take-All (WTA) fashion (see Figure 3.1). This allows the same neuron feature to be part of varying kinds of substructures through variable connectivity and allows for overlap without interference.

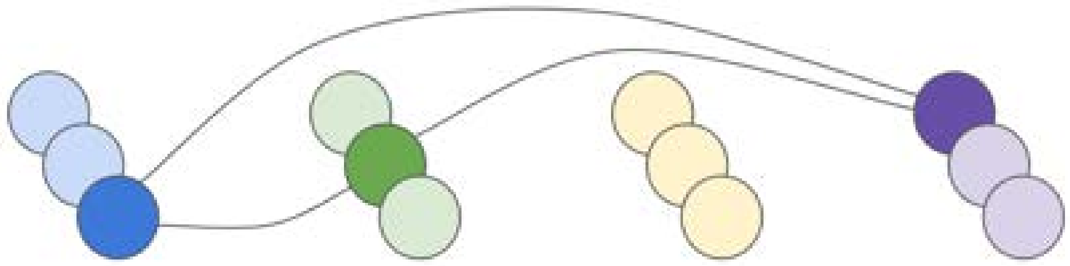

FIGURE 3.1: Example of four multi-cellular units, each containing three repeated neurons. In this configuration, the dark blue, green, and purple neurons are activated due to their connection strengths.

## 3.2 Forward Pass Learning

Typically, neural networks are trained by backpropagation of the error gradients through the layers in reverse direction. Since network self-organization does not rely on a target state that is to be converged onto, but rather the statistical properties of the input, we propose updates to our novel architecture to be performed during the forward pass in an unsupervised fashion. By comparing the sensory input and finding correlations the connections between cells and their strengths will be updated through a Hebbian learning process. In our particular case where the input consists of feature maps from a convolution layer, we can directly calculate pixel-wise dependencies and use this information to build the lateral connections. Since our LCL is a subset of the full network, this forward pass learning and backpropagation of errors directly influence each other and will have to be aligned carefully.

## 3.3 Prototype Description

The following section outlines the prototype architecture of our novel *laterally connected layer* (LCL), named after its main differentiating factor — the lateral, intra-layer connections. Figure 3.2 gives an overview of the relevant processing steps inside the LCL, though some steps are only performed during the training phase.

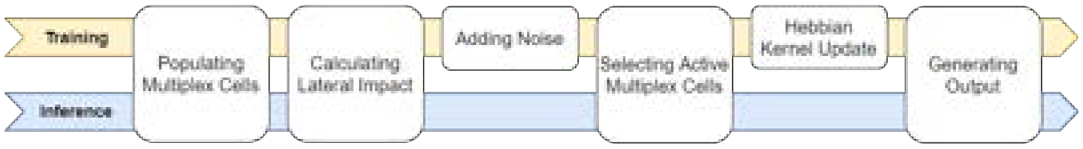
Training
Inference
Populating Multiplex Cells
Calculating Lateral Impact
Adding Noise
Selecting Active Multiplex Cells
Hebbian Kernel Update
Generating Output

FIGURE 3.2: Overview of the training and inference process of the laterally connected layer (LCL).

Our LCL implementation is based on two core concepts, the idea of connections between neurons of the same layer (referred to as *lateral connections* due to their orientation orthogonal to the information flow through the layers of a deep learning model) and allowing every neuron to be replicated several times, similar to the multi-cellular units in [52] (referred to as *multiplex units*). Lateral connections are represented through kernel filters, similar to the kernel filters of convolutional layers. That allows us to benefit from years of research in understanding and efficiently computing said filters, as well as fully utilizing current deep learning hardware (e.g. GPUs and TPUs). Furthermore, we automatically earn a degree of translational invariance where a pattern has not to be shown in every possible location. The kernel filters that govern the lateral connections are formed to represent statistical correlations between provided features and form the basis of a consistent network of information flow. Given the popularity of the Transformer architecture [35], we hypothesize that our lateral connections are comparable to the attention mechanism.

As mentioned in Section 3.1, any input given to this type of layer is expected to describe a set of two-dimensional feature maps. We expect the LCL to be adaptable for different kinds of features in the future but see it as an additional step outside the scope of this thesis. Furthermore, the activation function of the convolutional layer just before the LCL is exchanged for the hyperbolic tangent function, simplifying the numeric ranges that can be achieved by various calculations as part of the LCL.

### 3.3.1 Populating Multiplex Units

Multiplex units increase the space for possible lateral connections, creating additional possibilities to combine the same feature in completely different sub-networks. Because the connections are completely independent, a feature $A$ can have very little influence on feature $B$ in one configuration and completely dominate in combination with a different set of multiplex cells. In addition, it is biologically impossible for a neuron to constantly fire. While we do not strictly implement a refractory period for our neurons, spreading the work across cells reduces their individual duty cycle.

All $N$ identical repetitions of a feature (referred to as *multiplex cells*) form a *multiplex unit*, for which only one cell can finally be activated through the selection process described later in this section. A multiplex unit can be seen as a meta-neuron, which does not activate one of its cells for each and every input signal. It is thus perfectly acceptable that a multiplex unit chooses not to activate at all. Multiplex cells abstractly represent a neuron that activates given a feature, but in practice we represent each multiplex cell as a feature map activation. That means, each and every feature map given to the LCL as an input is represented through a multiplex unit, that repeats the feature map $N$ times to populate its multiplex cells.

### 3.3.2 Calculating the Lateral Impact

Before calculating the influence of lateral connections, we need to describe the kernel filters' desired behavior. Lateral kernel filters connect two multiplex cells from different units and represent a directional relationship between the source feature map and the target feature map. Applying the convolution onto the source feature map outputs the signal of the target feature map that should be strengthened.

Figure 3.4 shows how a kernel could look like, given that feature map $A$ and feature map $B$ are active together (in the Hebbian sense) and thus strengthen the pattern seen in target feature map $B$. In order to simplify the example, all cells marked with 1 are active and all empty cells are inactive (0). We now iterate over all pixel positions of feature map $B$ and compare the visible part of feature map $A$ relative to this position, remembering which pixels are activated relative to the current pixel to

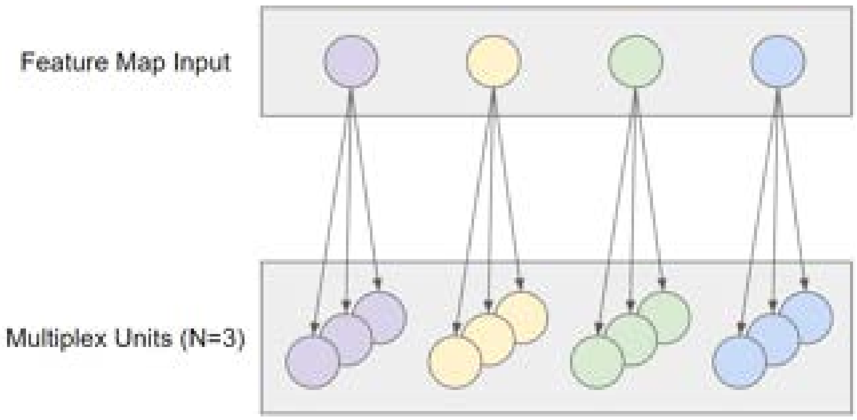

FIGURE 3.3: Example of populating four multiplex units with three cells each (lower part of the figure) from the input feature maps given to the LCL (top part of the figure).

form a partial kernel. The partial kernels are shown with a blue and orange header for the two corresponding pixels of feature map $B$. Summing up all partial kernels gives the final kernel, describing the lateral connection *from A to B*.

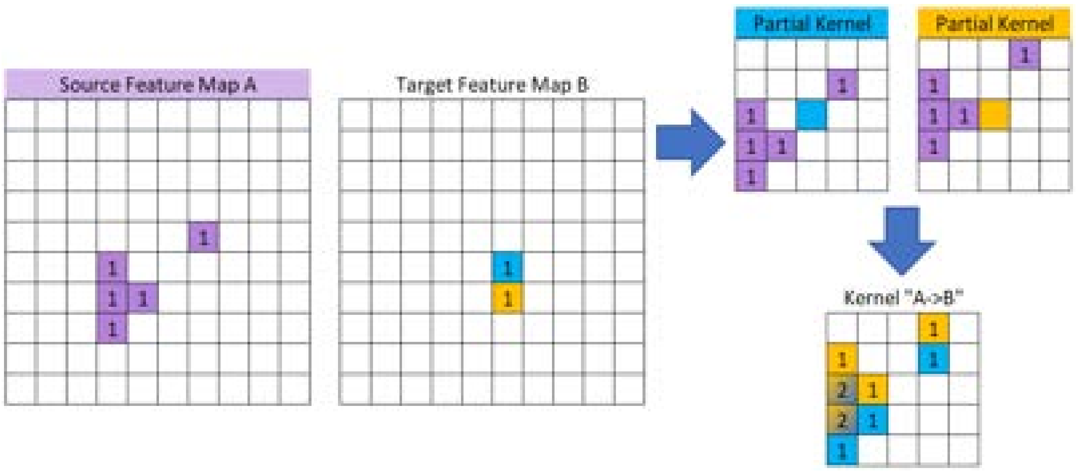

FIGURE 3.4: Example of a $5 \times 5$ lateral kernel that is formed to strengthen the pattern in feature map $B$ given the pattern of feature map $A$. The partial kernels show the pattern of feature map $A$ in relation to every strongly activated pixel in feature map $B$ (partial kernels), as well as the resulting final kernel "A->B".

Given the kernel, we should be able to reproduce the pattern of feature map $B$ just from feature map $A$ by performing a convolution using the lateral kernel "A->B". Figure 3.5 visualizes this convolution. On the left side of the figure, feature map $A$ is shown and in the middle is the kernel "A->B". When calculating the convolution operation, we get the output on the right side of the figure, which we refer to as the *lateral influence of feature map A on feature map B*. While a certain degree of blurriness is visible, we calculate a value of 0.7 at the pixel positions where feature map $B$ is activated, while all other positions reach at most 0.3. In this example we only use a single lateral connection, but our LCL stabilizes its pattern using a larger number of lateral influences and thus performs as an ensemble of different lateral influences.

The *lateral impact* is similar to the lateral influence, but instead of generating the

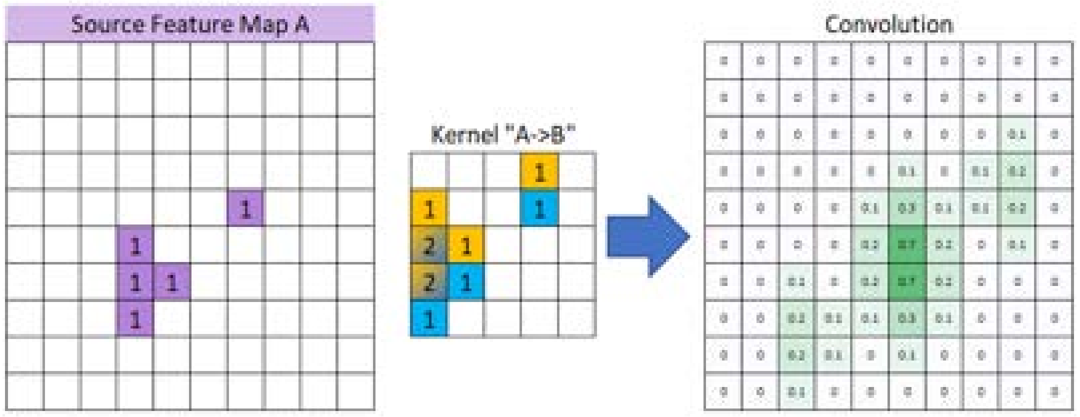

FIGURE 3.5: Example of applying the learned lateral kernel to feature map $A$. The resulting output of the convolution strongly correlates to the activated pixels in feature map $B$ (see Figure 3.4).

convolution as a feature map, we sum up all activations in the lateral influence[1]. This scalar value describes how strongly a feature map stabilizes another and is a valuable tool in comparing different multiplex cells inside the same multiplex unit.

Unlike in later steps where the convolution is only performed with lateral connections between active multiplex cells, the lateral impact is calculated for all available cell combinations, as it is at the foundation of the multiplex selection process.

### 3.3.3 Adding Noise

Repeating a feature multiple times will result in identical signals. Because of this, initial lateral signals are very close in strength because the lateral kernels have not had the time to specialize yet. Since only the active multiplex cells and their chosen connections are updated, it is vital that a certain degree of randomness is introduced to break the symmetry in every multiplex unit. Of course, the long term vision is that the multiplex selection process becomes deterministic and chooses to activate those cells which specialized on the given input signals. However, getting to this point takes time and an equal opportunity has to be given to all cells to specialize[2].

Random noise sampled from a uniform distribution causes all multiplex cells to be chosen at random for initial updates. By doing so, we artificially spread patterns across different combinations of active multiplex cells and ensure that a larger part of the available state space is used. The magnitude of the noise is controlled by the $\eta$ parameter. With increasing number of training iterations, the $\eta$ is slowly reduced to zero until the input signals fully deterministically decide which sets of multiplex cells are activated. At inference however, this step is skipped.

### 3.3.4 Selecting Active Multiplex Cells

After calculating the lateral impact of all possible connections and the addition of noise, we are able to start choosing for each multiplex unit, which cell is activated.

Figure 3.6 shows an overview of how the lateral connections calculating the lateral impact. Since all multiplex cells can be the source and target feature map of a

[1]The inspiration is drawn from the gram matrices used in the popular texture synthesis method by Gatys et al. [41], summarized in Section 2.4.2.

[2]This process has similarities in reinforcement learning, where models initially act randomly and slowly converge on acting according to the learned weights (exploration vs. exploitation).

lateral connection, we display every multiplex cell once in the left column as source and once on the right side as a target feature map. We only portray the connections for a single target feature map (marked as a yellow circle with a blue border). Dotted lines represent connections that are never used, due to the Winner-Take-All constraint on multiplex units. On either side of the feature maps, we added examples of what they would look like for an input image showing the digit 9.

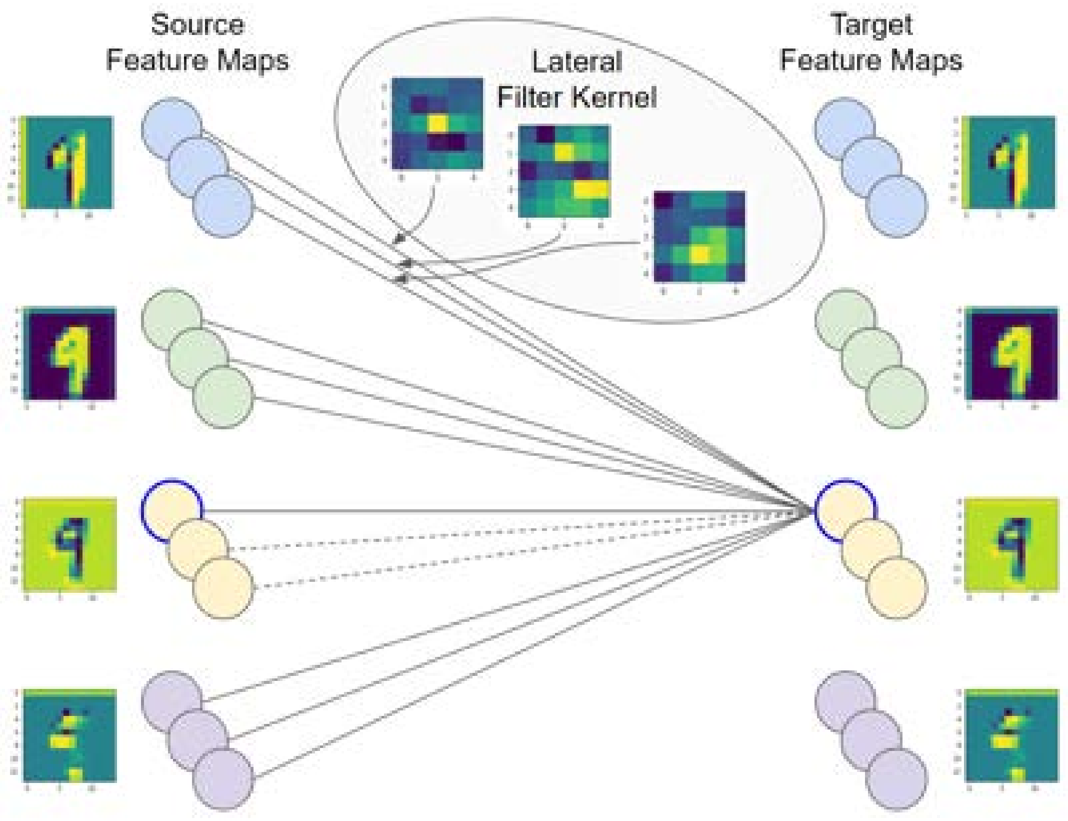

FIGURE 3.6: Visualization of the possible lateral influences between source feature maps (left column) and target feature maps (right column). For visual clarity, only the connections towards a single target feature map are shown. Every feature map is drawn both as source (left column) and target feature map (right column). Lateral connections are characterized by a filter kernel, shown for three connections. To give an example of the activations in these feature maps, the feature maps of the digit 9 in the MNIST dataset are shown next to every multiplex unit.

For every connection in Figure 3.6, we hold a $k \times k$ sized filter kernel (in our example, $k = 5$) that is applied to the source feature map to strengthen the learned pattern in the target feature map. The sum over the resulting feature map — the lateral impact — governs, which connections remain active and which ones are deactivated. For every multiplex unit, only the cell's connection with the highest lateral impact is kept and all others are set to zero. In Figure 3.7, we demonstrate the updated example from Figure 3.6 after removing all inactive connections.

### 3.3.5 Hebbian Update of Lateral Connections

The core of the LCL lies within the lateral connections, represented, and learned as convolutional kernel filters. Atypically though, the update does not depend on a

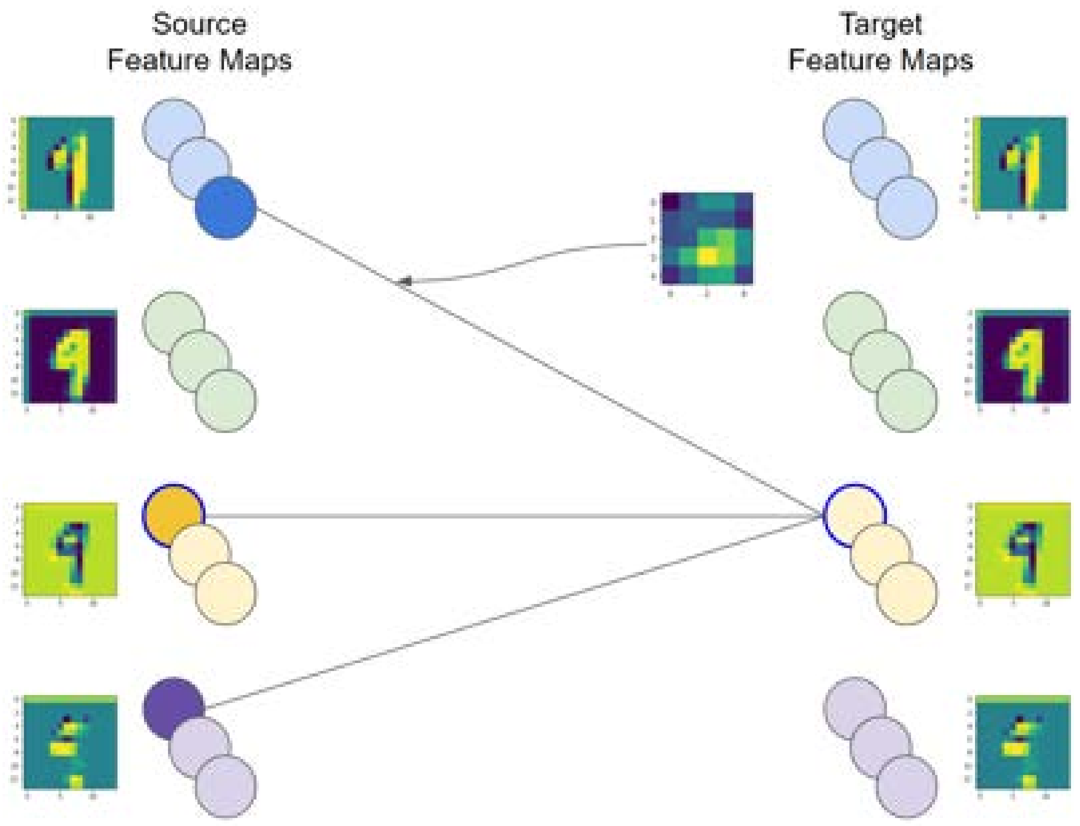

FIGURE 3.7: Visualization of selecting the most active lateral connections according to the lateral impact for a single target feature map. Compared to Figure 3.6, only the active connections are still shown. The selected multiplex cells in the source column are shown in bold colors. Note that not every multiplex unit (here green) needs to have an active connection to every target.

backpropagated error, but rather the simultaneous activation of two feature maps at the same position. The central pixel of the lateral kernel measures how often both feature maps (source and target) are active at the same time - similar to the value of the gram matrices in [41]. For all other pixels, the source feature map is shifted to match the relative position of the kernel pixel relative to the central pixel.

We call this update Hebbian, in the sense that when feature maps are both activated simultaneously in the same pixel locations, their lateral kernel value increases. If only one or none of them are active, their value decreases. By this process, we follow the Hebbian rule where cells that often activate together associate and strengthen their connection, while their connection strength decreases for all other cases. For the lateral multiplex cells to be active, it is thus not only necessary that the input must be right, but also the surrounding environment.

### The Lateral Kernel

More concretely, the lateral kernel $K$ is a four-dimensional structure containing the two-dimensional kernels between two multiplex cells (or feature maps). The first two dimensions describe the source and target cells, while the last two dimensions hold the height and width of a kernel. The size of the two-dimensional kernels is given by $k = 2 * d + 1$, where $d$ describes how large the distance between pixels can be at most to influence another pixel laterally and $F$ is the set of feature maps (including all $n$ repetitions). Following these instructions, we arrive at shape for $K$ of $[F_{Source}, F_{Target}, k_{Height}, k_{Width}]$. Given feature maps $a$ and $b \in F$ and pixel posititions $x \in k_{Height}$ and $y \in k_{Width}$, the top left pixel on the kernel defining the lateral connection "b -> a" is given by $K_{b,a,0,0}$. The directionality of these connections is important since they are not expected to behave symmetrically (and neither do so in practice).

Updating $K$ requires to iterate over all lateral connections from active multiplex cells and generate the pixel-wise product of the source and target feature map pairs, defined by the lateral connection. For kernel sizes $k$ larger than zero, the source feature maps must be shifted across to match the relative position of every pixel in the $[k \times k]$ lateral connection kernel. For example, with a $3 \times 3$ kernel, calculating the top left pixel requires the source feature maps to be shifted one pixel to the left and up, with the target feature map unshifted to calculate the diagonal influence.

Equation 3.1 shows how the changes to the lateral kernel are calculated using a Hebbian update. $A$ represents the activations given to the LCL after setting up the multiplex cell repetitions and has shape $[F, H, W]$, where $H$ and $W$ are the height and width of the feature maps, respectively. Please note that the activations are zero padded with a $d$-pixel wide border around the feature maps (i.e. the last two dimensions $H$ and $W$). The padding enables convolutions centered at border pixels with source and target feature map pairs translated relative to each other. We leave out the padding in Equation 3.1 in order to reduce the complexity of indexing $A$.

$$\Delta K_{b,a,x,y} = \frac{1}{HW} \sum_{i=0}^{H-1} \sum_{j=0}^{W-1} A^{Source}_{b,i-d+x,j-d+y} * A^{Target}_{a,i,j} \tag{3.1}$$

The changes $\Delta K$ are zero for all inactive lateral connections and all nonzero values are scaled to the range [0, 1], though a value of one would only be achieved if both feature maps are entirely made up of ones. To simplify the combinatorial possibilities for all numbers involved, we assume that $A \in [-1, 1]$ and $K \in [0, 1]$. This can for example be achieved by changing the activation function of the previous convolution layer to the tangens hyperbolicus function, instead of the in convolutional layers more commonly used ReLU function. In Equation 3.2, the calculated changes $\Delta K$ are updating the kernel $K$, with $t$ indicating the time step. The update is only performed on active lateral connections (see Section 3.3.4), otherwise the kernel values would further decrease over time and introduce a passive forgetting mechanism. $\alpha$ is the kernel learning rate, controlling how quickly changes are impacting the LCL. We expect this hyperparameter to behave similarly compared to learning rates encountered in other machine learning algorithms.

$$K^{t+1}_{b,a} = \begin{cases} (1-\alpha) * K^t_{b,a} + \alpha * \Delta K & \text{if lateral connection is selected} \\ K^t_{b,a} & \text{otherwise} \end{cases} \tag{3.2}$$

### 3.3.6 Generating Output from Active Multiplex Cells

The final output calculation is very similar to the process of calculating the lateral impact outlined in Section 3.3.2. However, we only use the lateral connections that survived the multiplex selection process of Section 3.3.4. The lateral influences for every active target multiplex cell are summed up to form the result. Because we choose one multiplex cell per multiplex unit, we return the same number of feature maps as are given to the LCL (i.e., every multiplex units yields one feature map).

The output of the LCL forms a small ensemble of multiplex cells, where active regions correspond to a consensus across all lateral influences of said feature map. Equations 3.3 and 3.4 demonstrate how the output $O$ is generated by calculating the convolution result $L$ of all lateral kernels and then is scaled to match the mean of the input activations $A$. We do not believe this scaling relative to the input is a long-term solution, but we found measurable improvements after starting to use it as the output would otherwise become too small for subsequent layers.

$$L_{a,x,y} = \sum_{b=0}^{|F|-1} Conv(A_b, K_{b,a}) = \sum_{b=0}^{|F|-1} \left( \sum_{i=0}^{k_H} \sum_{j=0}^{k_W} K_{b,a,i,j} * A_{b,x+i,y+j} \right) \tag{3.3}$$

for all active lateral connections $K_{b,a}$

$$O_{a,x,y} = L_{a,x,y} * \frac{\sum_a \sum_x \sum_y L_{a,x,y}}{\sum_b \sum_i \sum_j A_{b,i,j}} \tag{3.4}$$

# 4

# Experiments

## 4.1 Datasets

We chose the MNIST and MNIST-C datasets as they allow us to test our novel LCL architecture on a simple but well understood benchmark, while providing a large array of corruptions to evaluate the different representation learned by our layer.

### 4.1.1 MNIST

The MNIST dataset is a very famous benchmark of handwritten Arabic digits (zero through nine). The goal is to classify each image and assign the correct digit. The MNIST dataset contains 60′000 training and 10′000 test samples. All images are 28 by 28 pixels large, each digit is centered and all pixel values are black or white (corresponding to zero and one). Examples of such images are shown in Figure 4.1.

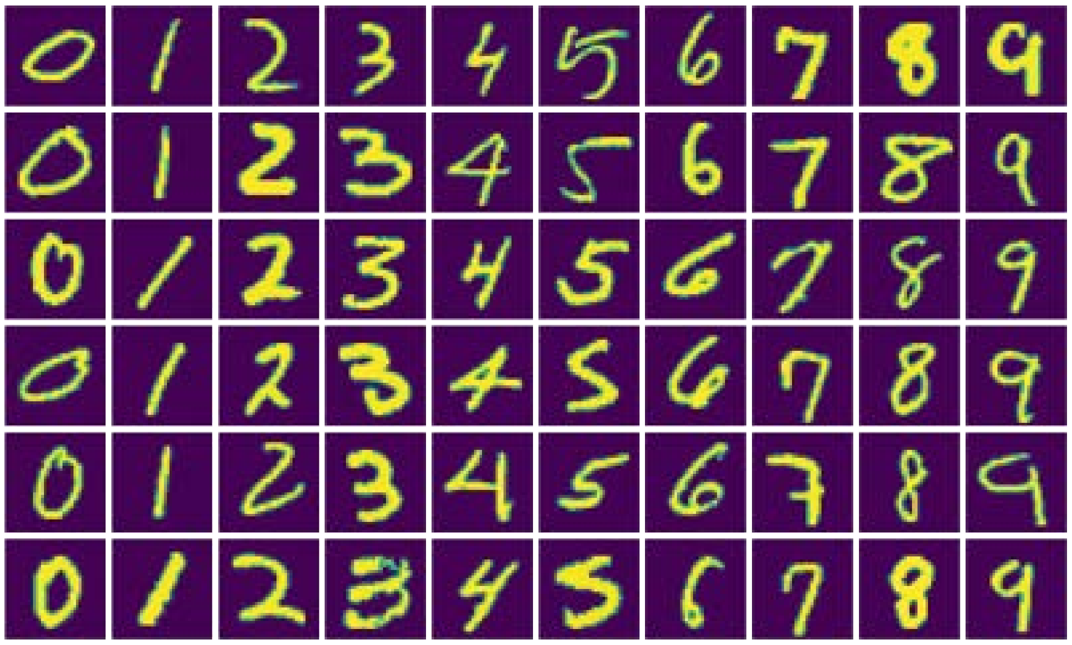

FIGURE 4.1: Examples from the MNIST dataset across all digits.

Because of the small size and ease of understanding, the majority of students and academics will have seen and played with this dataset in one form or another. It became famous through the success of LeCun et al. [22] over thirty years ago, when they demonstrated convolutional layers for the recognition of handwritten digits.

### 4.1.2 MNIST-C

The MNIST-C dataset [54] was created in order to evaluate the robustness of computer vision algorithms on a benchmark that is considered to be solved by the community[1]. However, the authors of the MNIST-C dataset argue that "models lack robustness to small translations of the input, small adversarial perturbations, as well as commonly occurring image corruptions such as brightness, fog and various forms of blurring" [54]. Examples of the corruptions in MNIST-C are shown in Figure 4.2.

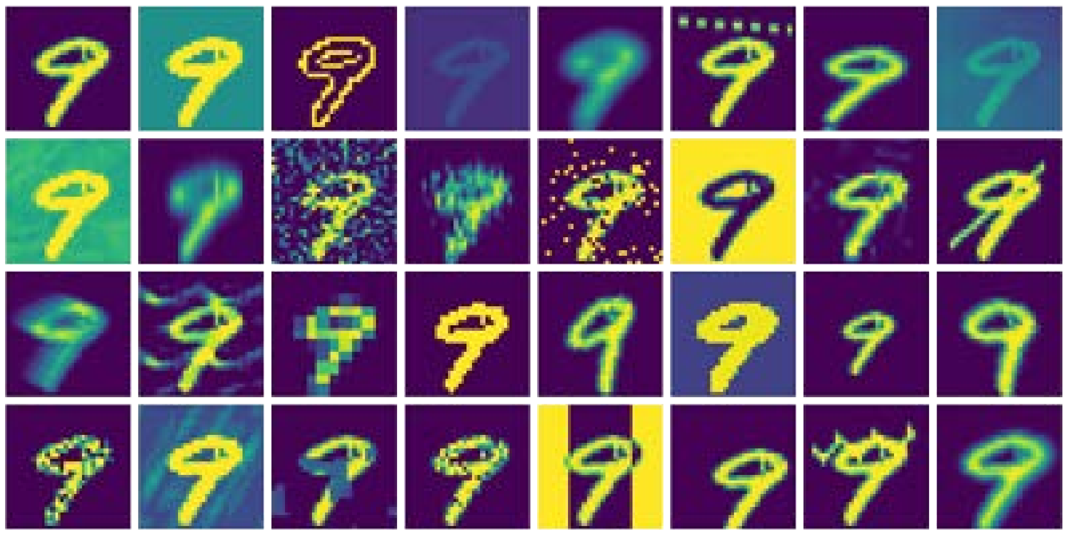
FIGURE 4.2: Examples of the 31 corrptions in the MNIST-C dataset. The top left sample shows the clean reference image from MNIST.

### 4.1.3 Relevance of MNIST-C Corruptions

The authors of the MNIST-C dataset used four principles to create the dataset:

- **Non-triviality**: The corruptions should have a significant impact on the performance of modern convolutional networks and not be simply overcome by what the authors call "naive data augmentation".
- **Semantic preservation**: Even though modern convolutional networks should be fooled, it remains important that humans are still able to recognize the depicted digits. The authors verified this claim by visual inspection.
- **Realism**: All corruptions should have a legitimate chance to be encountered in the real world. They attribute the corruptions to "perturbations to the camera setup, environmental factors, or physical modification".
- **Breadth**: Even though they published 31 variations of image corruptions, the authors only used a subset of 15 to conduct their evaluations as to reduce overlap in checking corruptions with the same kind of error.

The 31 corruptions provide a broad spectrum of challenges on which to test our novel LCL architecture. However, we do not expect the LCL to be able to cope with all corruptions equally. Depending on the receptive field (given by the previous

[1]Error rates have been shown to reach below 1% on MNIST [55] and an extended MNIST dataset (EMNIST) [56] has been released to increase the difficulty of the dataset.

convolutional layers and the size of the lateral kernels), especially global changes to the images will pose a potentially insurmountable challenge. By contrast, the LCL should perform well with small perturbances (e.g., locally applied noise).

For this reason, we categorize the corruptions in the following groups. Short summaries of what each corruption does are taken from [54]. The scripts to generate all corruptions can be found in their GitHub repository[2].

### Local Noise

Corruptions in this group only apply changes to the pixels accounting for the digit but leave the background unchanged. This fact alone reduces the difficult of the corruptions in this group, as the errors are clustered at the digit markings.

The main challenge in these corruptions is that the pixel values either do not reach the same magnitude due to blurring or the lines are not continuous with large differences between neighboring pixels. We expect the LCL to perform best (across all groups) on this particular set of corruptions, given that the lateral connections can strengthen very small discrepancies on the pixel level. Futhermore the LCL does not have to deal with random spurious signals in the background areas.

Figure 4.3 shows examples from the nine corruptions in this group. They are: **canny_edges** (applying the Canny edge detector), **defocus_blur** (replicating a camera out of focus), **gaussian_blur** (using Gaussian blur), **glass_blur** (recreating the view through glass), **motion_blur** (applying blur along a randomly chosen line), **pixelate** (scaling the image down and then back up), **shot_noise** (random corruptions from a Poisson distribution), **speckle_noise** (random corruptions from a normal distribution) and **zoom_blur** (changing the focal length during capture).

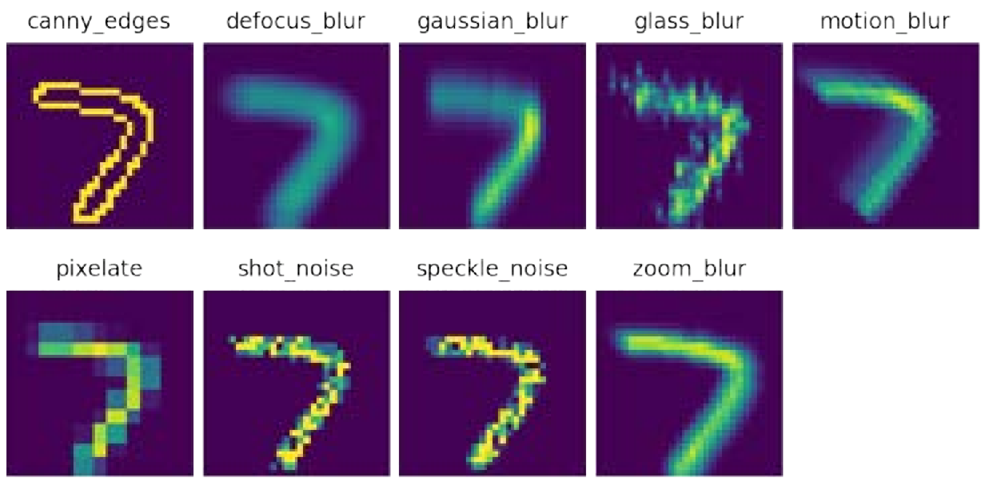

FIGURE 4.3: MNIST-C examples of the local noise group.

### Global Noise

Unlike the local noise group described above, the corruptions in this group not only affect the digit but are applied on the whole image. An argument could be made to put the spatter corruption into the group of superimposed objects. However, we

[2]https://github.com/google-research/mnist-c/blob/master/corruptions.py

believe the spatters to be more similar to noise, since they exhibit random patches rather than identical objects that are only translated and oriented differently.

A different argument could be made that the LCL is exposed to Gaussian noise during the training process. However, our added noise is applied to the lateral impact rather than the input of the whole network and does not directly impact the output of the LCL (but rather the multiplex selection process). We expect the LCL to struggle with these kinds of corruptions, as they require a large receptive field, which would result in a stronger low-pass (or blurring) effect on the produced feature maps. In addition, all the noisy activations in the background can stabilize random net fragments, rather than the ones activated by the digit sections alone.

In Figure 4.4, we show examples of the corruptions in this group. They are: **fog**, **frost** (added patterns of real frost), **jpeg_compression** (adds the artefacts of a lossy JPEG compression), **gaussian_noise** (applies Gaussian noise), **impulse_noise** (adds randomly replaces pixels with a constant value), **pessimal_noise** (adversarially trained noise), **snow** (simulates falling snow) and **spatter** (adds random patches). pessimal_noise was created by the dataset authors to adversarially add noise calibrated specifically to one of their models and is only kept for completeness.

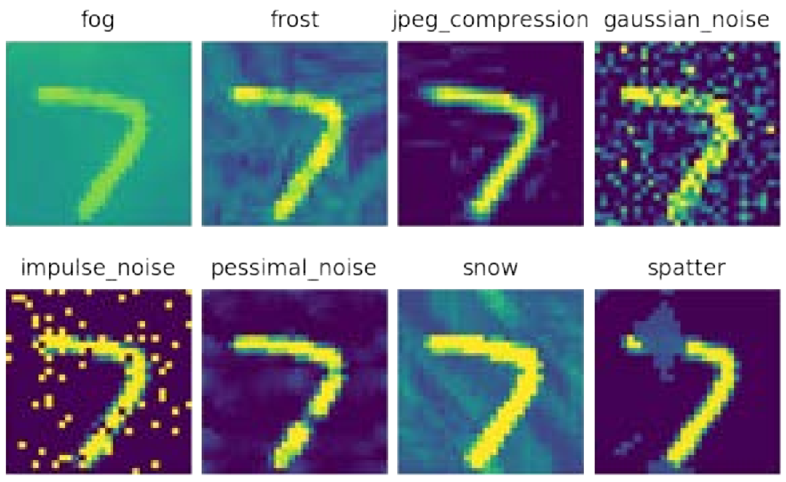

FIGURE 4.4: MNIST-C examples of the global noise group.

### Image Transformations

In this group, the corruptions alter the shape of the digit. Depending on the variance of the digits written in MNIST, there exists a good chance that many of the transformed digits were seen during training through a different handwriting. We believe the LCL has no mechanisms to perform significantly better than a typical convolutional layer as the statistical properties of the digits remains largely untouched.

The transformations in this group are: **elastic_transform** (applying an affine transformation), **rotate** (rotating around a randomly chosen point), **scale** (downscaling while preserving the aspect ratio), **shear** (displacing the digit with increasing magnitude along a line) and **translate** (shifting the digit to a different position).

### Superimposed Objects

This group contains objects of a fixed type (lines, dotted lines, and zigzag lines) which are placed on top of the image in random orientations and locations. This

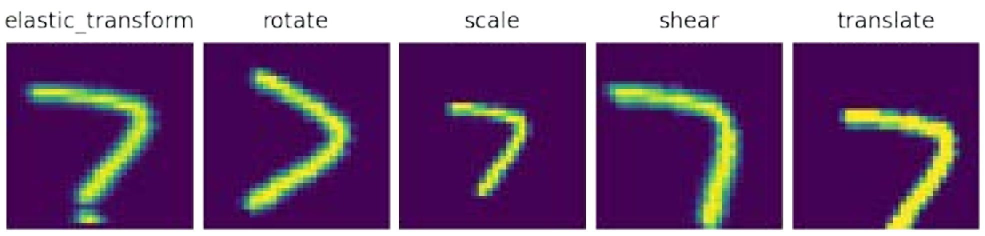

FIGURE 4.5: MNIST-C examples of the image transformations group.

group is very challenging, as there exist many possibilities to turn one digit into another by adding lines in specific spots (for example turning a one into a seven into a five, as can be seen in Figure 4.6). Correcting these corruptions likely requires a large receptive field. We believe the LCL will struggle with this kind of corruption, as it goes beyond denoising. Furthermore, the continuous lines share statistical properties with the lines used to draw the digits, making it hard to detect whether a line belongs to a digit or was randomly added by the corruption.

All three corruptions are shown in Figure 4.6. They are: **dotted_line**, **line** and **zigzag**. The main differences between the corruptions are the shape (straight vs. zigzag) and whether the lines are continuous or not (dotted_line vs. line).

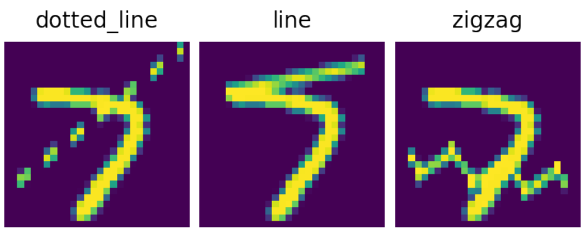

FIGURE 4.6: MNIST-C examples of the superimposed objects group.

### Global Image Settings

In this group, all corruptions change the global image statistics (e.g., brightness and contrast), that could be reversed by applying pre-processing step that take into account the average brightness or contrast. Since this group always applies their transformations on the whole image (or in the case of stripe a large majority), we believe the LCL will be affected comparable to a convolutional network.

The six corruptions in this group are shown in Figure 4.7. They are: **brightness** (increasing the brightness of the whole image), **contrast** (reducing the contrast across the whole image), **inverse** (inverting all pixel values), **quantize** (reducing the color range by rounding pixel values), **saturate** (increasing the saturation) and **stripe** (inverting the pixels in the left and right most quarter of the image).

## 4.2 Model Overview

In this section, we describe the models used in our experiments. In order to measure the effectiveness of our LCL architecture, we compare between a small convolutional neural network and the identical network with an added LCL.

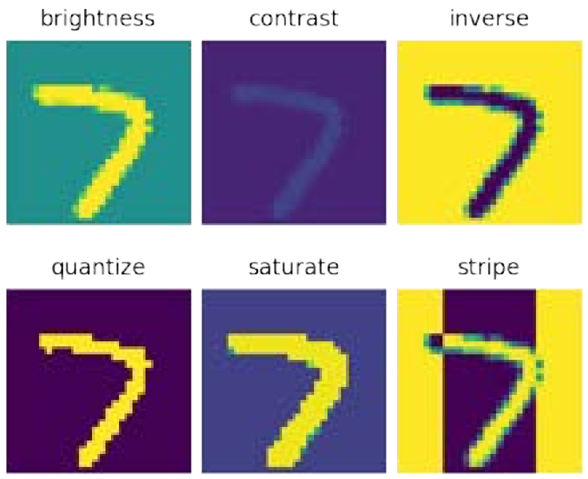

FIGURE 4.7: MNIST-C examples of the global image settings group.

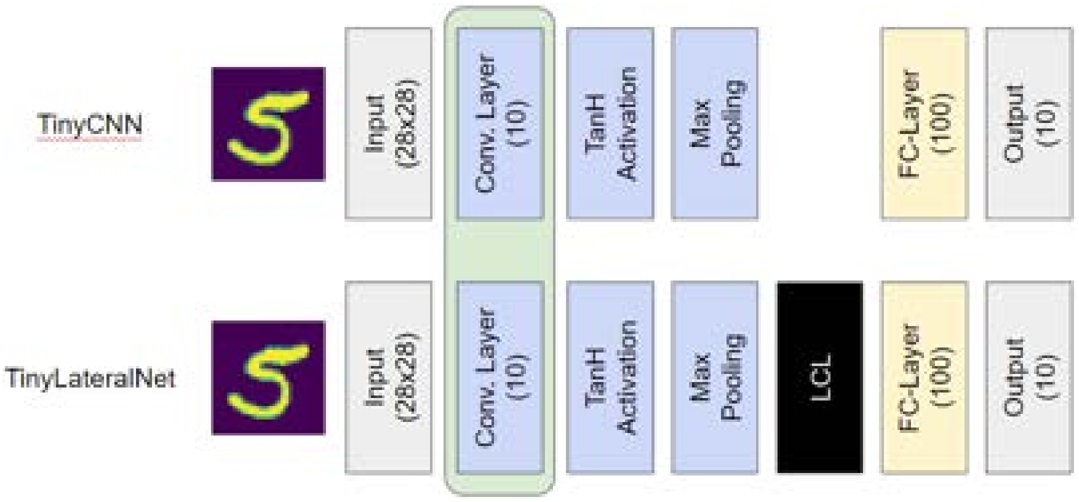

FIGURE 4.8: Schematic view of the models used in our experiments.

### 4.2.1 TinyCNN

The TinyCNN is designed to represent a very small convolutional neural network (hence the name). Input images are given to a convolutional layer with ten feature maps, followed by the tangens hyperbolicus activation[3]. A max pooling layer further increases the receptive field of the network. The output of the pooling layer is flattened and fed to a fully connected layer with 100 neurons. Finally, the output layer assigns probabilities for the ten classes of digits found in the MNIST dataset.

We differentiate between the fully trained (FT) version, where the TinyCNN is trained completely from scratch, and the pre-trained (PT) version, where the convolutional layer is initialized with the best model across all fully trained TinyCNNs. When the convolutional layer is pre-trained, its weights are frozen and only the remaining part of the model (the fully connected layer) is trained. Figure 4.8 depicts and overview of the TinyCNN's building blocks. Marked in green is the convolutional layer, for which the weights are frozen in the pre-trained variant.

[3]A numerical simplification compared to ReLU, for more details see Section 3.3.5.

### 4.2.2 TinyLateralNet

The TinyLateralNet is a copy of the TinyCNN, where the only difference is an added laterally connected layer before the fully connected layer. Analogous to the TinyCNN, we also differentiate between fully trained (FT) and pre-trained (PT) versions, where the convolutional layer's weights are taken from the best fully trained TinyCNN in the latter version (the same model checkpoint as for the pre-trained TinyCNN). Figure 4.8 shows an overview of the TinyLateralNet components.

## 4.3 Experimental Setup

In our experiments, we use the official training and test sets of MNIST[4] and the static MNIST-C dataset[5]. The training set consists of 60'000 samples with 6'000 samples per digit class, the training set contains 10'000 samples with 1'000 per digit. We use $\frac{5}{6}$ of the training set for training and $\frac{1}{6}$ as the validation set. All models were trained using early stopping, terminating after no improvement on the validation loss for three epochs and using the model weights at the lowest validation loss.

The magnitude of the added noise in the LCL is kept at $\eta$ for half an epoch before it is reduced linearly to zero towards the end of the first epoch. The lateral kernel is initialized to weights drawn from a uniform distribution $\in [0, 1)$ and then multiplied by 0.02. This value has been empirically found[6] to be smaller than the mean value of change introduced to the lateral kernel by $\Delta K$. In theory, this allows the kernel to start in a non-zero state but increase its magnitude over the initial state across time. However, given that the lateral connections are expected to organize themselves sparsely and converge to several activated patterns, the observed mean both for the lateral kernel and the changes applied to it quickly decrease across the first epoch (during which noise is added). After this period, they have reached an equilibrium state, where the statistical properties remain constant.

The code has been written in Python version 3.8.10 using the PyTorch environment for efficient, scalable execution on the GPU. Experiments were conducted on an NVidia GTX 1080Ti with CUDA using the dependencies provided in our GitHub repository[7]. A detailed overview of the code base can be found in the Appendix B.1.

## 4.4 Discussion on LCL-specific Hyperparameters

The laterally connected layer (LCL) presented in the previous chapter introduces a variety of new hyperparameters that ought to be tuned. In this section we outline their impact on the training of the LCL and propose ranges to fit their interactions.

The **number of multiplex cells** $n$ per multiplex unit controls how much room is given to the lateral connections for spreading the subnetwork structures across the layer. We hypothesize that increasing this parameter will have diminishing returns as only as much space has to be provided for all necessary patterns to be stored. Further increasing $n$ beyond this point should have no effect.

[4] `http://yann.lecun.com/exdb/mnist/`

[5] `https://zenodo.org/record/3239543#.YsD4M3ZBxhE`

[6] We encourage the interested reader to take a look at the Python notebooks provided in our GitHub repository: `https://github.com/edualc/mt_object_recognition/tree/master/notebooks`

[7] `https://github.com/edualc/mt_object_recognition/blob/master/requirements.txt`

The **size of the lateral kernels** $k$ (or $d$, given that $k = 2 * d + 1$) governs the size of the neighborhood which can laterally influence any given pixel. Because the proposed kernel structure does not inherently differentiate the lateral influence proportional to the distance of the central pixel, we expect to see an increasingly visible blurring effect (or low pass filter) on the produced output. Additionally, the number of parameters (kernel weights) of the LCL grows quadratically with the size of the kernel. Because of these reasons, the value of $k$ should be kept small (for example by stacking multiple LCLs).

The **lateral learning rate** $\alpha$ governs how drastic the changes to the kernel weights are. Given that we do not want to erase previously learned patterns and that during training a large number of update steps will be performed, we believe it makes sense to choose small values for $\alpha$. However, the LCL is to some extent modelled after the associative abilities of i.e., Hopfield networks and their update in the original binary form is performed as a concrete update step where the learning rate would be one. Given this analogy, we will explore up to the upper limit of one but expect optimal values to lie much closer to zero.

The **noise magnitude** $\eta$ has to be set relative to the kernel changes $\Delta K$ and the lateral impact, such that a enough multiplex cells are switched away to explore otherwise unchosen cells. However, we still want a significant part of the kernel adapting to the statistical patterns seen during the training. When the noise is eventually reduced, the space of multiplex cells should have formed meaningful lateral connections to deterministically active those net fragments that are responding to the current input. Similar to the lateral learning rate, we also believe an optimal value to be rather small. The overall signals of the lateral influence are scaled to the range of $[0, 1]$, but likely never reach that high, since that would mean both the kernel and the input are mostly made up ones.

Summarizing this list, we summarize our proposed ranges for the hyperparameter search in Table 4.1 below. The ranges have been iterated upon through the lifetime of this thesis and we have converged empirically to the listed ranges below. We refer the interested reader to Section B.2 in the Appendix, where we outline additional information about the process involved in learning about the inner workings of the LCL architecture and confirm the theoretically expected behavior.

TABLE 4.1: Proposed search space for LCL hyperparameters.

| Hyperparameter | Range | Search Space |
|---|---:|---|
| Number of multiplex cells $n$ | [2, 8] | Discrete |
| Lateral distance $d$ | {0, 1, 2} | Discrete |
| Lateral learning rate $\alpha$ | [1e-4, 1e-0] | Continuous |
| Noise magnitude $\eta$ | [1e-3, 5e-1] | Continuous |

### 4.4.1 Hyperparameter Optimization

To find appropriate hyperparameters, we performed a large scale hyperparameter search using the Optuna[8] library. Optuna allowed us to automatically search through the vast space of potential hyperparameters and do so efficiently.

[8] `https://optuna.readthedocs.io/en/stable/index.html`

Since our goal was to keep both TinyCNN and TinyLateralNet as similar as possible, we did not investigate different number of feature maps in the convolutional layer or changing its kernel size. These values were fixed at ten feature maps and $3 \times 3$-sized kernel filters. The only parameter left to be tuned is the learning rate.

For the TinyLateralNet, the number of tuneable hyperparameters is larger (as shown in the previous section). Given that the search space scales exponentially with the number of hyperparameters, we increased the number of trial runs for the TinyLayerNet, but only linearly. That means for every $T$ trials runs of the TinyCNN, we ran a multiple of $T$ trials runs on the TinyLateralNet. While this gives an advantage to the TinyCNN, we believe that the learning rate of the TinyCNN has a larger impact on the overall performance compared to every individual hyperparameter of the TinyLateralNet and is thus acceptable. Furthermore, we chose the number of trials $T$ to be 100, giving a large sample size for both models.

### TinyCNN Hyperparameter Search

The hyperparameter search for the TinyCNN is much simpler compared to the TinyLateralNet, because only the learning rate needs to be adjusted. We quickly found an optimal value for the fully trained TinyCNN at $4e-4$ and $2e-4$ for the pre-trained version. Given that the pre-trained variant already has well-adjusted filter kernels in the convoluational layer, it is not surprising that the learning rate is lower.

### Initial Hyperparameter Search for the TinyLateralNet

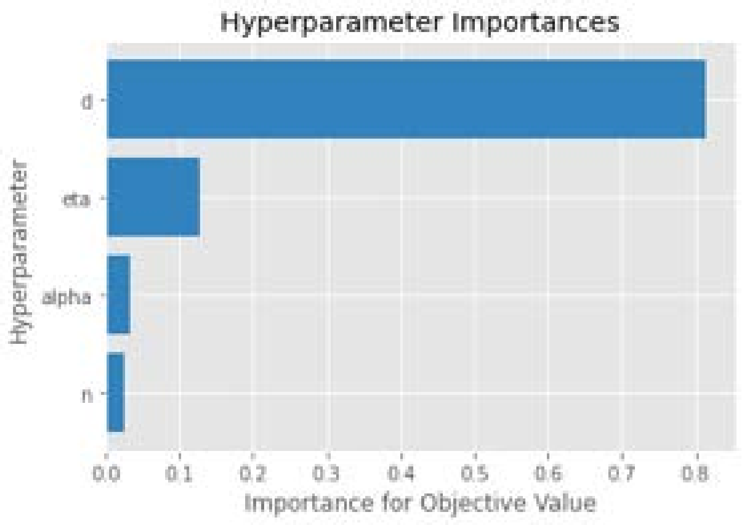

FIGURE 4.9: Parameter importance of the TinyLateralNet given by Optuna for a fixed learning rate of $3e-4$. The objective value is the validation loss on MNIST.

Figure 4.9 shows the hyperparameter importance for a fixed learning rate of $3e-4$ in 400 trials. It is clearly visible that the size of the lateral kernel $k$ given by $k = 2 * d + 1$ has the most impact on the overall performance, followed by the magnitude of the noise added by $eta$ and both the lateral kernel learning rate $alpha$ and the number of multiplex cells $n$ barely impact the performance. A parallel coordinate plot of the distance $d$ can be seen in Figure 4.10. In this figure, we clearly see that the larger kernel negatively impacts the model's performance by up to 1% accuracy. Note that not all 400 trials are shown in this figure, but only the best 40 trial runs.

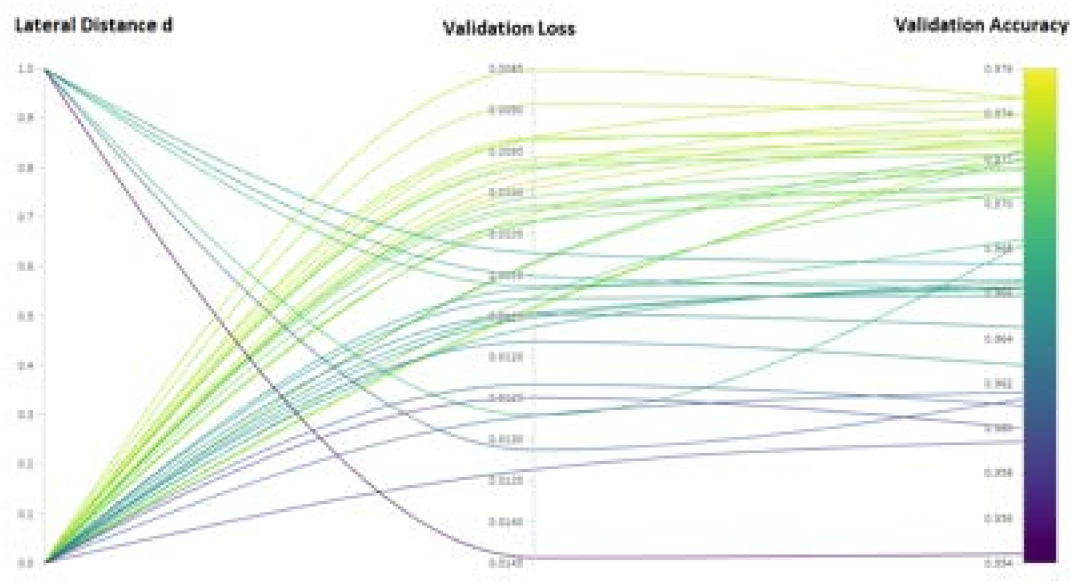

FIGURE 4.10: Parallel coordinate plot of the distance parameter $d$, the validation loss and accuracy on MNIST across 400 trials on the TinyLateralNet (only the best trials are visualized).

From this hyperparameter search we chose the distance $d$ to be set to zero, resulting in $1 \times 1$-sized lateral kernels (i.e., without influence from neighboring pixels).

### TinyLateralNet (Fully Trained) Hyperparameter Search

The hyperparameters that dominate the TinyLateralNet training are the learning rate and the magnitude of the noise $\eta$, as shown in Figure 4.11. We hypothesize that the lateral learning rate $\alpha$ has a lower importance, as the number of iterations during the training process is large enough to iterate over and store all required patterns. Similarly, the importance of the number of multiplex cells $n$ points towards enough space being available even for lower values. If there was not enough space to save all necessary pattern combinations, the overall performance of the TinyLateralNet would increase significantly, which we did not observe in our hyperparameter search.

### TinyLateralNet (Pre-Trained) Hyperparameter Search

Figure 4.12 depicts the hyperparameter importance for the TinyLateralNet with pre-trained convolutional weights. Compared to the fully trained variant, the learning rate is much less important. Given that a significant part of the network starts in a well calibrated setting, it is not surprising that $\alpha$ and $\eta$ see an increase in importance instead. $n$, however, is equally unimportant and shows that the MNIST dataset is simple enough to be represented even with few multiplex cell repetitions.

### Final Hyperparameters

The final values of our hyperparameter search are given in Table 4.2.

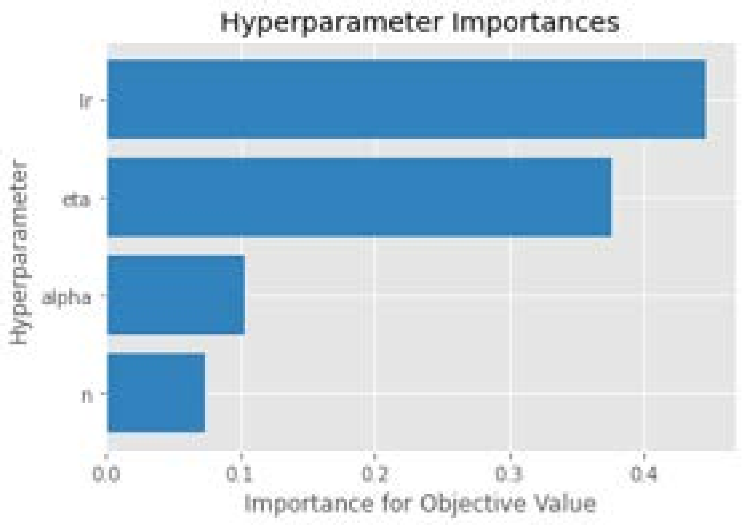

FIGURE 4.11: Parameter importance of the fully trained TinyLateral-Net with a kernel size of 1. The objective value is the validation loss on MNIST.

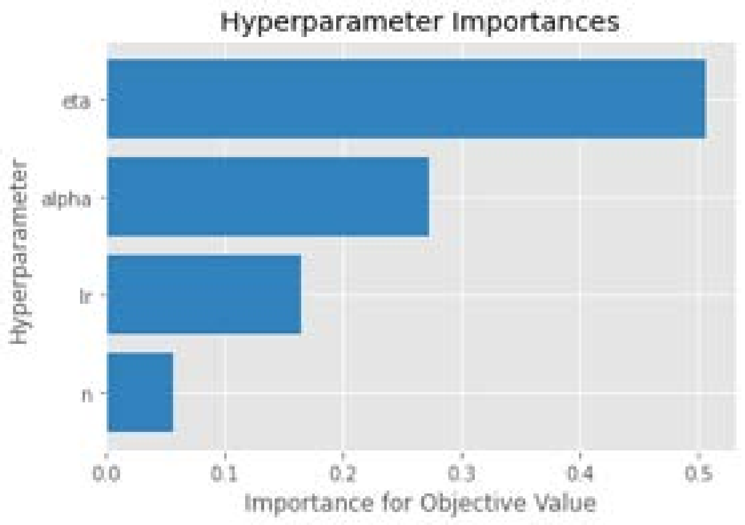

FIGURE 4.12: Parameter importance of the pre-trained TinyLateral-Net with a kernel size of 1. The objective value is the validation loss on MNIST.

TABLE 4.2: Final hyperparameters per model type

| Hyperparameter | TinyCNN (TCNN) | | TinyLateralNet (TLN) | | |
|---|---|---|---|---|---|
| | Fully Trained | Pre-Trained | Fully Trained | | Pre-Trained |
| Learning Rate | 4e-4 | 2e-4 | 3.6e-4 | 3.9e-4 | 2e-4 |
| Noise Magnitude $\eta$ | - | - | 0.05 | 0.031 | 0.25 |
| Lateral Learning Rate $\alpha$ | - | - | 0.0012 | 0.11 | 0.05 |
| Number of Multiplex Cells **n** | - | - | 5 | 3 | 3 |
| Lateral Distance **d** | - | - | 0 | 1 | 0 |

TABLE 4.3: Experiment results on the TinyCNN and TinyLateralNet.

| **Dataset** | **CG†** | **TinyCNN (TCNN)** | | **TinyLateralNet (TLN)** | | |
|---|---|---|---|---|---|---|
| | | Fully Trained | Pre-Trained | Fully Trained (d=0) | Fully Trained (d=1) | Pre-trained (d=0) |
| MNIST test set | - | $^{\bullet}_{*}$0.9828 ± 0.0017 | $^{\bullet}$**0.9840** ± 0.0008 | 0.9730 ± 0.0047 | 0.9543 ± 0.0339 | 0.9745 ± 0.0019 |
| canny_edges | LN | 0.4488 ± 0.0391 | $^{\bullet}$0.3905 ± 0.0446 | $^{\bullet}$**0.6471** ± 0.1494 | $^{\bullet}$0.5942 ± 0.1570 | 0.3168 ± 0.0259 |
| defocus_blur | LN | 0.5768 ± 0.0770 | $^{\bullet}$0.4314 ± 0.0319 | $^{\bullet}$0.7884 ± 0.0540 | $^{\bullet}$**0.8038** ± 0.1661 | 0.3226 ± 0.0181 |
| gaussian_blur | LN | 0.6517 ± 0.0581 | $^{\bullet}$0.5606 ± 0.0171 | $^{\bullet}$**0.8393** ± 0.0451 | 0.7640 ± 0.2959 | 0.4687 ± 0.0094 |
| glass_blur | LN | 0.8600 ± 0.0296 | 0.7617 ± 0.0153 | $^{\bullet}$**0.9114** ± 0.0168 | $^{\bullet}$0.9040 ± 0.0263 | $^{\bullet}$0.8164 ± 0.0095 |
| motion_blur | LN | $_{*}$0.8403 ± 0.0382 | $^{\bullet}$0.7413 ± 0.0217 | **0.8428** ± 0.0409 | 0.7447 ± 0.0963 | 0.6817 ± 0.0415 |
| pixelate | LN | **0.9385** ± 0.0090 | 0.9182 ± 0.0045 | 0.9349 ± 0.0166 | 0.9304 ± 0.0358 | 0.9144 ± 0.0076 |
| shot_noise | LN | 0.8806 ± 0.0299 | $^{\bullet}$0.7551 ± 0.0194 | $^{\bullet}$**0.9627** ± 0.0068 | $^{\bullet}$0.9303 ± 0.0387 | 0.6663 ± 0.0289 |
| speckle_noise | LN | 0.9107 ± 0.0218 | $^{\bullet}$0.8172 ± 0.0149 | $^{\bullet}$**0.9653** ± 0.0066 | 0.9378 ± 0.0371 | 0.7388 ± 0.0232 |
| zoom_blur | LN | $_{*}$**0.9300** ± 0.0182 | $^{\bullet}$0.9086 ± 0.0100 | 0.9186 ± 0.0165 | 0.8561 ± 0.1011 | 0.8438 ± 0.0193 |
| fog | GN | $_{*}$0.4518 ± 0.0757 | 0.2702 ± 0.0491 | **0.4532** ± 0.0952 | 0.1921 ± 0.1250 | 0.2690 ± 0.0397 |
| frost | GN | $_{*}$0.4168 ± 0.0604 | $^{\bullet}$0.2939 ± 0.0340 | **0.5141** ± 0.2115 | 0.2110 ± 0.1746 | 0.0478 ± 0.0282 |
| jpeg_compression | GN | $^{\bullet}_{*}$0.9774 ± 0.0021 | $^{\bullet}$**0.9781** ± 0.0005 | 0.9690 ± 0.0039 | 0.9477 ± 0.0296 | 0.9414 ± 0.0052 |
| gaussian_noise | GN | 0.3705 ± 0.0597 | $^{\bullet}$0.1124 ± 0.0088 | 0.4479 ± 0.2269 | $^{\bullet}$**0.5885** ± 0.2634 | 0.1025 ± 0.0107 |
| impulse_noise | GN | 0.4601 ± 0.0460 | $^{\bullet}$0.1746 ± 0.0293 | 0.4418 ± 0.1844 | $^{\bullet}$**0.7108** ± 0.2835 | 0.1171 ± 0.0166 |
| pessimal_noise | GN | $^{\bullet}_{*}$**0.8842** ± 0.0292 | $^{\bullet}$0.8088 ± 0.0109 | 0.8072 ± 0.0771 | 0.8290 ± 0.0529 | 0.6851 ± 0.0210 |
| snow | GN | $_{*}$0.6012 ± 0.0702 | $^{\bullet}$0.3785 ± 0.0201 | **0.6517** ± 0.2296 | 0.3115 ± 0.2058 | 0.0609 ± 0.0225 |
| spatter | GN | $^{\bullet}_{*}$**0.9620** ± 0.0025 | $^{\bullet}$0.9587 ± 0.0016 | 0.9438 ± 0.0040 | 0.8971 ± 0.0218 | 0.7781 ± 0.0163 |
| elastic_transform | IT | $^{\bullet}_{*}$0.7544 ± 0.0160 | $^{\bullet}$**0.7551** ± 0.0080 | 0.7049 ± 0.0296 | 0.6764 ± 0.0891 | 0.7061 ± 0.0185 |
| rotate | IT | $^{\bullet}_{*}$0.8668 ± 0.0077 | $^{\bullet}$**0.8751** ± 0.0051 | 0.8384 ± 0.0159 | 0.8047 ± 0.0578 | 0.8434 ± 0.0084 |
| scale | IT | $^{\bullet}_{*}$0.7269 ± 0.0432 | $^{\bullet}$**0.7719** ± 0.0129 | 0.5831 ± 0.1069 | 0.5310 ± 0.1326 | 0.6726 ± 0.0654 |
| shear | IT | $^{\bullet}_{*}$**0.9418** ± 0.0030 | $^{\bullet}$0.9396 ± 0.0033 | 0.9192 ± 0.0145 | 0.8789 ± 0.0610 | 0.9159 ± 0.0087 |
| translate | IT | $^{\bullet}_{*}$0.3755 ± 0.0205 | $^{\bullet}$**0.4056** ± 0.0118 | 0.3006 ± 0.0344 | 0.3020 ± 0.1075 | 0.3662 ± 0.0167 |
| dotted_line | SO | $^{\bullet}$**0.8722** ± 0.0169 | $^{\bullet}$0.8151 ± 0.0079 | 0.7984 ± 0.0434 | 0.8414 ± 0.0767 | 0.6673 ± 0.0234 |
| line | SO | $^{\bullet}_{*}$**0.8033** ± 0.0167 | $^{\bullet}$0.7714 ± 0.0045 | 0.7379 ± 0.0505 | 0.7088 ± 0.0305 | 0.6109 ± 0.0170 |
| zigzag | SO | $^{\bullet}_{*}$**0.7244** ± 0.0155 | $^{\bullet}$0.6797 ± 0.0086 | 0.5492 ± 0.0567 | 0.6248 ± 0.0867 | 0.5132 ± 0.0296 |
| brightness | GIS | $_{*}$0.4076 ± 0.0705 | $^{\bullet}$0.3005 ± 0.0445 | $^{\bullet}$**0.6019** ± 0.2697 | 0.2056 ± 0.1840 | 0.0800 ± 0.0314 |
| contrast | GIS | 0.1308 ± 0.0223 | $^{\bullet}$0.1283 ± 0.0096 | 0.2051 ± 0.1105 | $^{\bullet}$**0.4003** ± 0.1247 | 0.1158 ± 0.0029 |
| inverse | GIS | 0.0604 ± 0.0277 | 0.1223 ± 0.0268 | $^{\bullet}$0.0957 ± 0.0295 | 0.0736 ± 0.0408 | **0.1370** ± 0.0301 |
| quantize | GIS | 0.9643 ± 0.0086 | $^{\bullet}$0.9489 ± 0.0028 | **0.9670** ± 0.0064 | 0.9464 ± 0.0348 | 0.8380 ± 0.0151 |
| saturate | GIS | $_{*}$**0.8327** ± 0.0344 | $^{\bullet}$0.6131 ± 0.0349 | 0.7906 ± 0.0952 | 0.4960 ± 0.2279 | 0.3680 ± 0.0402 |
| stripe | GIS | $_{*}$**0.5916** ± 0.1545 | $^{\bullet}$0.3494 ± 0.0327 | 0.5412 ± 0.2456 | 0.2374 ± 0.1904 | 0.0320 ± 0.0381 |

(Shown dataset performance is measured as accuracy on the test set ± one standard deviation.)

† CG refers to the grouping of MNIST-C corruptions outlined in Section 4.1.3. The abbreviations used are: Local Noise (LN), Global Noise (GN), Image Transformations (IT), Superimposed Objects (SO) and Global Image Settings (GIS).

$^{\bullet}$ Marked cells indicate a statistically significant improvement over the corresponding other model (with identical training, e.g. TinyCNN Pre-trained compared to TinyLateralNet Pre-trained). In the case of the fully trained TCNN, we use $^{\bullet}$ to signify significance compared to the fully trained TLN using $d = 0$ and $_{*}$ in comparison with TLN using $d = 1$. The significance has been evaluated using Welsch's T-test with $p < 0.05$.

## 4.5 Results

In this section, we analyze the performance of the TinyCNN (TCNN) and TinyLateralNet (TLN) on MNIST and the corruptions in MNIST-C. Table 4.3 shows the performance of the models on all datasets and corruptions. For a visual overview, we refer the interested reader to Figures B.7 and B.8, put in the Appendix for brevity.

The numbers shown in both Table 4.3 and Figures B.7 and B.8 display the accuracy from eight models trained with the hyperparameters described in Section 4.4.1. The value after the $\pm$ sign indicates one standard deviation. Bold numbers indicate the best accuracy of any model on the same dataset. The prefix $^{\bullet}$ signifies a statistically significant improvement over the other architecture with the same training (e.g. TCNN fully trained vs. TLN fully trained). Because the LCL architecture puts an emphasis on potential influences from neighboring pixels on the presented feature maps, we show both experiments with lateral distance $d \in \{0, 1\}$.

### 4.5.1 Network Comparisons Between Fully Trained and Pre-trained

Since all models were trained using only the clean MNIST dataset, we start by comparing the performance on the MNIST test set. For the purpose of this comparison, we will look at the TLN with $d = 0$, that is, without influence from neighboring pixels, given its superior performance given its lower standard deviation on the MNIST test set. Both TCNN models significantly outperform the TLN counterparts by 1%. Both the TCNN and TLN reach a comparable performance when comparing fully training the network and starting with a pre-trained convolutional layer. Both architectures achieve a higher median accuracy when using the pre-trained convolutional layer and a smaller standard deviation, though this could be expected, as the pre-trained weights already converged in the training of the model checkpoint used. However, a look at the MNIST-C variants paints a drastically different picture.

Using a pre-trained TCNN results in significant performance reductions of more than 20% points of accuracy on a variety of MNIST-C datasets. While a comparable performance on the MNIST test set can be achieved, the network is not trained explicitly to deal with image corruptions. We believe this causes a reduction in the generalization ability of the network, as the number of learnable network weights is reduced to only the neurons of the fully connected classifier.

The pre-trained TLN's accuracy plummets on a handful of datasets to below 15% accuracy (frost, gaussian_noise, impulse_noise, snow, brightness, contrast, and stripe), even when the fully trained variant performs two or more times better. These results indicate that the feature maps given to the LCL need to be calibrated differently compared to a regular CNN. Even though the pre-trained TLN reports a higher accuracy on six datasets (MNIST test set, elastic_transform, rotate, scale, translate and inverse), the difference is only significant on scale, translate and inverse, where the performance is already subpar with 67.26%, 36.62% and 13.7%, respectively.

As indicated by the $^{\bullet}$ markings in Table 4.3, the difference between the pre-trained TCNN and TLN grows large enough that the TCNN significantly outperforms the TLN on all but four datasets. In contrast, the pre-trained TLN is only a significant improvement over the TCNN on the glass_blur dataset.

### 4.5.2 A Question of Robustness

Given that adding the LCL to a convolutional network yields inferior results, our main question still remains: *Can we increase the robustness by accepting a decrease in* *overall accuracy?* To answer this question, we will now only compare the fully trained variants of both models, choosing the TLN without neighborhood influences ($d = 0$).

### Local Noise (LN) Group

We hypothesized that this group would benefit our LCL architecture the most, because relevant signals are arranged locally, and the background does not contain random signals. As expected, we can observe a statistically significant accuracy difference across six of the nine datasets (canny_edges, defocus_blur, gaussian_blur, glass_blur, shot_noise and speckle_noise), where the TLN achieves a median accuracy up to 21% points higher than the TCNN. For the remaining three datasets (motion_blur, pixelate and zoom_blur), we observe comparable results for both model types, with differences inside the margin of error. These results show that for the specific use case of locally arranged corruptions where global knowledge is not required to overcome the corruption, the TLN is able to match or outperform the TCNN.

### Global Noise (GN) Group

Evaluating the global noise group, the median accuracy of both models drops by up to 60 points compared to the MNIST test set. On fog, frost, gaussian_noise and impulse_noise datasets, both models struggle to recognize digits. On jpeg_compression and spatter, where the corruption impacts the whole image, but are more focused to small areas, both models remain above 95% accuracy, though the TCNN significantly outperforms the TLN. We believe introducing noise of global scale requires the TLN to have a much larger receptive field to be competitive.

### Image Transformations (IT) Group

The TLN is no match for the TCNN on the datasets of this group. $^{\bullet}$ markers show a significant dominance across all the corruptions, though both models show a similar relative degradation. However, there is a visible difference between corruptions that partially overlap due to the differences in handwriting (rotate and shear), compared to corruptions with a lower overlap probability (elastic_transform and scale) or completely out of distribution corruptions (translate). In particular, the MNIST dataset was created by centering the $20 \times 20$ large digits in the $28 \times 28$-sized image with a 4-pixel wide band of empty pixels on the outside. Because of this, any corruptions where the digit is moved into this boundary has a much larger impact compared to rotating the digit (where different handwritings easily differ in rotation[9]).

### Superimposed Objects (SO) Group

Among the superimposed objects, the dotted_line has the least impact on the performance. The digits are presented using continuous lines, rendering the dots in the dotted_line corruptions at least conceptually different from the strokes that truly represent the digit. Even so, the TCNN outperforms the TLN by 8% at 87.88% versus 79.84% and with a three times smaller standard deviation of about 1.5%. Between dotted_line and line, the performance degradation is larger for the TCNN, narrowing the performance gap to 7%. zigzag lines with their distinctive corners pose an even larger challenge, further decreasing the performance of both models. In this

[9]My handwriting for example is tilted forward by about 30 degrees.

group, the TCNN significantly outperforms the TLN on all datasets and by a good margin. In addition, the standard deviation is up to three times larger for the TLN.

#### Global Image Settings (GIS) Group

The last group features the largest discrepancy in accuracy across all its variants, where quantize is in the top three of least affected corruption variants across all models (fully and pre-trained) with 96.43% accuracy for the TCNN and 96.70% for the TLN, while also featuring the most impactful corruption, inverse, with 6.04% and 9.57% accuracy, respectively. Even though both TCNN and TLN are facing difficult challenges with all but the quantize corruption, the differences between the TCNN and TLN remain largely within the margin of error with the exception of brightness and inverse, where the TLN significantly outperforms the TCNN.

### 4.5.3 Increasing the Lateral Kernel Size

We will now take a look at the difference between a fully trained TLN with no neighborhood influences ($d = 0$) and a fully trained TLN with 1-pixel radius influences ($d = 1$). As shown in our hyperparameter search in Section 4.4.1 and Figure 4.9, the lateral distance $d$ is the most impactful hyperparameter for tuning. As presented in Table 4.3, the fully trained TLN with $d = 0$ reaches a higher median accuracy on the MNIST test set with 97.30% versus 95.43% with $d = 1$. At a six times larger standard deviation, there exists a run among the eight trained models with $d = 1$ that reaches 98.21% accuracy, which is higher than all TLN with $d = 0$ and is close to comparable to the TCNN models. However, this configuration is much less stable during training and prone to diverge catastrophically during training, with runs as low as 89.25%, diverging into an unsuitable state during the first three epochs.

Given that the performance of the TLN with $d = 1$ is diluted by runs that did not train until successful completion, it is not a surprise that there are much larger measured standard deviations compared to the TLN with $d = 0$. However, four datasets stand out: gaussian_noise, impulse_noise, dotted_line and contrast.

In the global noise group (GN), the accuracy of the TLN model with $d = 1$ decreases by 50% or more on fog, frost, and snow. These corruptions all affect the images with modest changes to surrounding pixels, but rather introduce broad patterns across the whole image that are strongly correlated throughout neighboring pixels. Compared to gaussian_noise and impulse_noise, pixels are corrupted independently of their neighbors, which the LCL architecture is able to pick up and mitigate, resulting in a significant improvement over the TCNN.

Among the superimposed objects (SO) group, the TLN with $d = 1$ outperforms the TCNN for some runs classifying dotted_line digits. The small pixel errors introduced by the dotted lines feature statistical characteristics that a $3 \times 3$ kernel is able to process, but a $1 \times 1$ kernel failed. The previously significant advantage of the TCNN over the TLN with $d = 0$ is no longer significant for the TLN with $d = 1$.

All models on contrast reach an accuracy of 20% of below, except for the TLN with $d = 1$ at 40.05%. While the reduced minimum and maximum ranges between strongly activated and deactivated regions in the feature maps are drastically reduced on the input, the larger LCL kernel is able outperform its competitors.

# 5 Conclusion and Future Work

## 5.1 Conclusion

In this thesis, we set out to find a more robust representation learned by a perceptron-based layer. We presented and evaluated a novel architecture based on lateral, intra-layer connections that are trained using Hebbian learning. We demonstrated significant benefits on groups of image corruptions, in particular where noise is concentrated to local regions or when adjacent pixels are disturbed independently.

While we currently must accept a performance reduction on the main task, we believe our work is the first step in learning representations that take the best ideas from both neuroscience and machine learning and yield more robust results. There remains still much to be done, but we believe our findings take a step out of the comfort zone of established machine learning credos and into unexplored territory.

Recently, discussions about the sentience of Google's chatbot LaMDA [57] have sparked conversations on the scientific social media channels about where artificial intelligence is headed. Hinton, one of the grandfathers of AI, is quoted saying he is "deeply suspicious" of backpropagation [58]. Meanwhile, LeCun — another grandfather of AI — released a paper draft [59] outlining a cognitive architecture that should bring us closer to human intelligence, including world models. What we gather from those discussions is that scaling up — as demonstrated so successfully by language models — will likely only bring us closer to human intelligence, but not surpass it. To paraphrase Marcus [60], scaling up the length of a ladder will bring us higher up, but a ladder is unlikely to bring us to the moon. As such we believe our findings could make a step in finding ladder alternatives for the future.

## 5.2 Discussion

Our experiments in Chapter 4 on MNIST and MNIST-C demonstrate, that our novel LCL architecture provides a representation capable of dealing with some, but not all types of corruptions introduced in MNIST-C. Under real circumstances, we would argue that a 1% drop in accuracy (as measured on the MNIST test set) can be worthwhile, given the significant advantage on a subset of corruptions.

From our results, we conclude that the LCL architecture shows promising potential and implements in practice, what we designed it to do in theory. We started by looking for a way to learn a more robust representation and were successful, but not yet in all possible parameters and without trade-offs. There remains a solid amount of future work to be done, as we outline in the next section.

In particular, our experiments were conducted on very small networks that are closer to a proof of concept, rather than networks applied in real-world use cases.

Similarly, the MNIST dataset has been considered solved for years and we look forward to upscaling the LCL architecture both in terms of scope and challenge.

## 5.3 Future Work

### 5.3.1 Limitations of the LCL architecture

We note a need for future work, given by the limitations that we observed. In our view, the following areas have a high potential impact on the LCL architecture:

**Dependence on clean training data**: Because the LCL kernel learn statistical dependencies in the local neighborhood regions of the given feature maps, it is important that the feature maps are processed without any artifacts, like the border effect caused by zero padding (see Section B.1.2 in the Appendix).

**Training instability**: Compared to the TinyCNN, we observed erratic behavior in many of the trained TinyLateralNet. The instabilities look to be correlated with the size of the lateral kernel $k$ and need to be further investigated.

**Low-pass filter**: With larger lateral kernels, the output of the LCL is visibly blurred as if a low-pass filter was applied. This phenomenon is to some extent expected, given that the convolutions with the lateral kernel are calculated using the neighborhood region of a pixel (which scales with kernel size), rather than the pixel alone. We believe further changes are necessary, to decouple this effect from the purpose of the pattern stabilization of the lateral kernel. CNNs are often structured with blocks convolutional layers that are followed by a pooling layer. We would propose to locate the LCL in front of the pooling layer, counteracting any low-pass filtering by the pooling operation.

**Numerical limitation to LCL input**: We limited the LCL to expect input feature maps to remain within the range of $[0, 1]$ to ease calculations of the Hebbian kernel update. However, we believe this limitation should be overcome to allow for example the more commonly used ReLU activation function to be applied to the convolutional layer before the LCL and reduce the changes necessary for incorporating a LCL into any neural network in the future.

**Computational efficiency**: Currently, the LCL is memory hungry, as it scales with the number of multiplex cells $n$, the number of incoming (and thus outgoing) feature maps $F$ and the size of the lateral kernel $k$, all quadratically in complexity class $O(n^2k^2F^2)$. In addition, the calculation of the Hebbian kernel update with shifted feature maps was sped up drastically using Einstein sum notations[1] in PyTorch, but scales in execution time quadratically with the lateral kernel size $k$. Both memory and speed can likely be improved upon — and is highly probable to be required — for use in larger neural networks.

### 5.3.2 Outlook

The following list is not directly dependent on the limitations of the LCL but look at broader applications of the presented architecture.

[1]See `https://pytorch.org/docs/stable/generated/torch.einsum.html` for the documentation.

**Forward learning vs. backpropagation**: During training, the whole network is updated with backpropagation except for the LCL. We believe this interaction between forward learning and backpropagation must be researched more thoroughly, as there needs to be a balance between how the convolution input of the LCL is trained and what the output of the LCL gravitates towards. Our suggestions would be to alternate the training direction every $t$ time steps to keep the training targets more consistent and investigating the demonstrated supervised forward pass learning [51] mentioned in Section 2.5.

**Incorporate additional concepts**: Our LCL is based primarily on the works of von der Malsburg [52, 53], but does not fully implement every aspect. Area of interest include scaling mechanisms for individual multiplex cells, building hierarchical representations through stacked layers and distributing the usage of multiplex cells across the available space (e.g., sparsity constraint).

**Corruption types**: The current LCL architecture primarily excels for a subset of local or independent noise corruptions. We believe the LCL should be able to cope with many more, given the right setup (i.e., larger kernels, stacked layers and a receptive field that spans the majority of the input image).

**Segmentation for padding and the border effect**: See Section B.1.2 on the issues around the border artefacts, introduced by the padding of previous convolution layers (as well as our convolution step). This works well in our setting, but for a general (noisy) case, a more sophisticated approach needs to be taken into consideration. A solution could be to use image segmentation to single out objects in an image. By doing so, a new image without noisy background could be generated that more closely resembles the images in our dataset.

# Appendix A

# Calculations

## A.1 Hopfield Calculations

### A.1.1 Hopfield Network Update

$$\sigma_i^{(t+1)} = sgn\left[\sum_\mu \left(F(\xi_i^\mu + \sum_{j\neq i}\xi_j^\mu\sigma_j) - F(-\xi_i^\mu + \sum_{j\neq i}\xi_j^\mu\sigma_j)\right)\right] \tag{A.1}$$

$$= sgn\left[\sum_\mu \left(F(\xi_i^\mu + \sum_{j\neq i}\xi_j^\mu\sigma_j) - F(\sum_{j\neq i}\xi_j^\mu\sigma_j - \xi_i^\mu)\right)\right] \tag{A.2}$$

### A.1.2 Hopfield Network Update for Quadratic Polynomials

In this section, we will show the equations when choosing $F(z) = z^2$. The *sign* function is omitted for brevity and the $\xi$'s belong to the $\mu$ of the outer most sum. Because the $\xi$ in Equation A.5 belong to two different indices ($i$ and $j$), connections are always between pairs of cells.

$$\sigma_i^{(t+1)} = \sum_\mu \left((\xi_i + \sum_{j\neq i}\xi_j\sigma_j)^2 - (\sum_{j\neq i}\xi_j\sigma_j - \xi_i)^2\right) \tag{A.3}$$

$$= \sum_\mu \left(\left[\xi_i^2 + 2\xi_i(\sum_{j\neq i}\xi_j\sigma_j) + (\sum_{j\neq i}\xi_j\sigma_j)^2\right] - \left[(\sum_{j\neq i}\xi_j\sigma_j)^2 - 2\xi_i(\sum_{j\neq i}\xi_j\sigma_j) + \xi_i^2\right]\right) \tag{A.4}$$

$$= \sum_\mu \left(4\xi_i(\sum_{j\neq i}\xi_j\sigma_j)\right) = 4\sum_\mu\sum_{j\neq i}\xi_i\xi_j\sigma_j \tag{A.5}$$

### A.1.3 Hopfield Network Update for Cubic Polynomials

In this section, we will show the equations when choosing $F(z) = z^3$. The *sign* function is omitted for brevity and the $\xi$'s belong to the $\mu$ of the outer most sum. Analogous to Equation A.5, from Equation A.8 follows that connections with higher order polynomials ($n > 2$) force connections to be between $n$-sized tuples of neurons (in this example $n = 3$ with indices $i, j, k$).

$$\sigma_i^{(t+1)} = \sum_\mu \left((\xi_i + \sum_{j\neq i}\xi_j\sigma_j)^3 - (\sum_{j\neq i}\xi_j\sigma_j - \xi_i)^3\right) \tag{A.6}$$

$$\begin{aligned}
&= \sum_{\mu}\Bigg( \left[ \xi_i^3 + 3\xi_i^2(\sum_{j\neq i}\xi_j\sigma_j) + 3\xi_i(\sum_{j\neq i}\xi_j\sigma_j)^2 + (\sum_{j\neq i}\xi_j\sigma_j)^3 \right] \\
&\quad - \left[ (\sum_{j\neq i}\xi_j\sigma_j)^3 - 3\xi_i(\sum_{j\neq i}\xi_j\sigma_j)^2 + 3\xi_i^2(\sum_{j\neq i}\xi_j\sigma_j) - \xi_i^3 \right] \Bigg)
\end{aligned} \tag{A.7}$$

$$= \sum_{\mu}\left( 2\xi_i^3 + 6\xi_i(\sum_{j\neq i}\xi_j\sigma_j)^2 \right) = 2\xi_i \sum_{\mu}\left( \xi_i^2 + 3(\sum_{j\neq i}\xi_j\sigma_j)^2 \right) \tag{A.8}$$

The squared sum of Equation A.8 should be resolved as:

$$\left(\sum_i a_i\right)^2 = \sum_i a_i^2 + 2\sum_{i<j} a_i a_j \Rightarrow \left(\sum_{j\neq i}\xi_j\sigma_j\right)^2 = \sum_{j\neq i}(\xi_j\sigma_j)^2 + 2\sum_{j\neq i,k\neq i,j<k}\xi_j\xi_k\sigma_j\sigma_k \tag{A.9}$$

### A.1.4 Hopfield Network Update for Exponential Functions

In this section, we will show the equations when choosing $F(z) = e^x$. The case for the exponential function can only be solved by replacing $e^x$ with its infinite expansion (the Maclaurin series) as follows:

$$e^x = \sum_{n=0}^{\infty}\frac{x^n}{n!} = \frac{x^0}{0!} + \frac{x^1}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + ... = 1 + \frac{x}{1} + \frac{x^2}{2} + \frac{x^3}{6} + ... \tag{A.10}$$

As such, *all* stored patterns (up to the choice of $n$) need to be known to calculate the $\sigma_i^{(t-1)}$ values. Because this is an infinite expansion, this energy function pushes the tuple requirement to theoretically include all neurons.

# Appendix B

# Additional Content

## B.1 Code

All code is available in our repositories on GitHub. The lion's share of this thesis' implementation can be found in the `mt_object_recognition`[1] repository. An additional repository `mt_image_generation`[2] contains an implementation base on the popular Unity[3] game engine to generate image datasets of arbitrary 3d objects.

### B.1.1 Lateral Connections Package

The code found in the `mt_object_recognition` GitHub repository includes all code used for the experiments of this thesis and features the implementation of the laterally connected layer (LCL). The necessary information to install and run the dependencies can be found in the `README.md` file, located in the base directory. Additionally, the base directory contains many of the python scripts used to run experiments or longer running tasks, often remotely on a GPU cluster. The provided `Dockerfile` allows you to create a Docker container with all the necessary dependencies to run the code remotely. The other directories are explained below in more detail:

`_old`

This directory contains experimental, but finally unused code, such as part of the implementation of the "Texture Synthesis using CNN" paper [41] and older iPython notebooks. We do not recommend spending further time here.

`experiment_results`

This directory hosts the many CSV files from our experiments with class predictions and results across the various datasets (primarily MNIST-C).

`images`

A few samples images, as well as all datasets can be found here. Note that the datasets are not directly provided in the GitHub repository, but rather expected to be downloaded and put into this directory. For MNIST, you can use the `torchvision` library to automatically download it for you (into the `mnist` directory). For MNIST-C, please use the static dataset provided by the paper authors on their GitHub repository[4] and store it in the `mnist_c` directory. For example, the MNIST-C variant *dotted_line* is expected to be found at path `mt_object_recognition/images/mnist_c/dotted_line/train_images.npy`.

[1] `https://github.com/edualc/mt_object_recognition`
[2] `https://github.com/edualc/mt_image_generation`
[3] `https://unity.com/`
[4] `https://github.com/google-research/mnist-c`

`lateral_connections`

The main implementation code for the LCL is found in this directory. The `character_models.py` file hosts a variety of models with and without using the LCL. `dataset.py` implements different datasets that were largely used for debugging purposes, but not in the most recent version of ablation studies. `lateral_model.py` implements an older variant of an integrated LCL, but is not used anymore. `layers.py` contains the LCL implementation, of which there are three variants[5]. Please use `LaterallyConnectedLayer3`, as it is the most recent version and implements the architecture described in this thesis. Versions one and two implement an inferior set of characteristics, such as a scaling mechanism and multiplex selection on a per-pixel basis (rather than per feature map). The `loaders.py` file implements utility methods for quickly loading the datasets as PyTorch loaders. Similarly, `model_factory.py` implements methods to quickly load a model with default configurations and even loads the checkpoint weights, if a path is provided. `torch_utils.py` features a variety of methods that were used to apply min-max scaling and softmax on a feature map level, rather than across the whole data cube.

`models`

All the network checkpoints are stored in this folder. This also includes pre-trained checkpoints that were used for fine-tuning or re-training the networks with an added LCL. It is currently not planned that we upload our model checkpoints, as they can easily be recreated locally with the training scripts.

`notebooks`

All the iPython notebooks are stored here, most of which should be visible through GitHub already (though just rendered without interactivity in mind.) All notebooks with the `Ablation__` prefix include ablation studies and debugging scenarios to test, whether the LCL performs as it should.

`scripts`

Here you can find helper scripts to quickly activate the Python environment, start Jupyter or check the GPU usage. `dgx_get_gpu.sh` is an example call for running this code on the ZHAW GPU cluster infrastructure.

The experiments of this thesis outlined in Chapter 4 using the hyperparameter search with Optuna can be recreated using the scripts found in the base directory and are marked with the prefix `optuna_`. Any future changes to the repository will be marked in the `README.md` file, please check here first in case of differences.

### B.1.2 Padding / Border Effect Issues

For CNNs, it has become the norm to use to so called `same` padding, where every feature map receives and additional $\frac{kernel\_size-1}{2}$ pixels on every side. This is done to keep the feature map sizes constant across multiple convolution steps and to allow each pixel on the border of a feature map to interact with the same number of kernel positions as any pixel in the feature map. The values of the added pixels can differ, often they are zero. The VGG19 network that we planned to use applies `same` padding with zeros [42], and so do the TinyCNN and the TinyLateralNet.

[5] `LaterallyConnectedLayer`, `LaterallyConnectedLayer2` and `LaterallyConnectedLayer3`

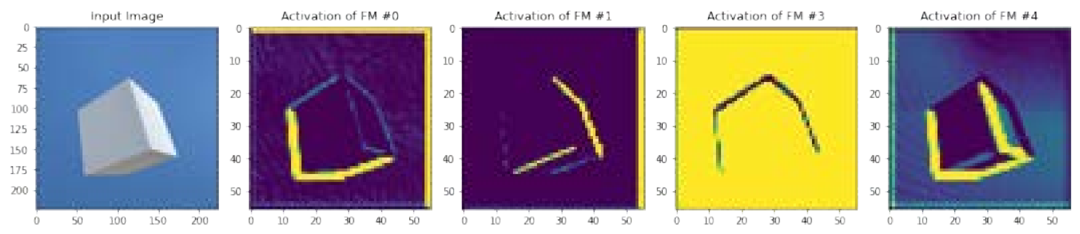

FIGURE B.1: The input image and a selection of four feature map activations of VGG19, demonstrating unusual activity at the edges.

However, as presented by [61], the choice of padding has a significant impact on the performance of the CNN model. As shown in Figure B.1, the feature maps produced by VGG19 depict unusual activity at the edges that do not intuitively match the kind of edge detection patterns otherwise observed in the activations.

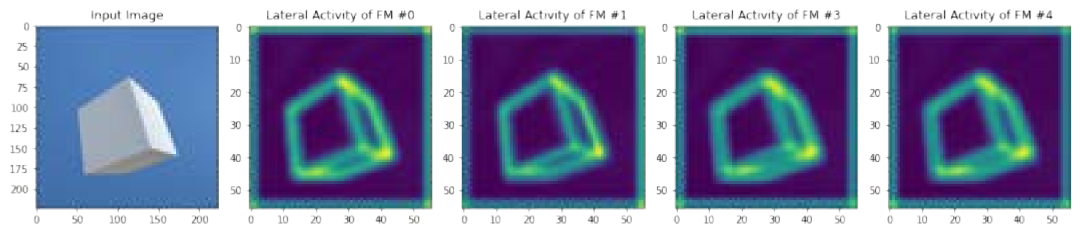

FIGURE B.2: The input image and lateral activites of our model.

The goal of our lateral model is to learn from the statistical correlation between local features. Because the artifacts at the edge of the feature maps are not caused by the content of an input image, but zero value padding, the edges are a strong feature pattern which stabilizes itself through the training iterations, as shown in Figure B.2. While the outlines of the cube are clearly visible, all around the edges (and especially in the corner pixels), we can observe a strong lateral activity.

In order to repair these artifacts as a source for our network, we remove the content of the border edges relative to the padding and kernel size of VGG19's last convolution layer that we use. In addition, we increased the radius even further, to also catch the propagation of this effect through layered applications of multiple convolution steps. For the `Pool2` pooling layer with a kernel size of $3 \times 3$ and padding of 1 pixel, we would ordinarily remove $\frac{kernel\_size-1}{2}$ pixels on all sides. However, because there are two convolutional layers between the pooling layers, we use 2 pixels. Missing values are filled in through symmetrical padding, which has been shown to exhibit comparable foveation properties to not padding feature maps at all [61].

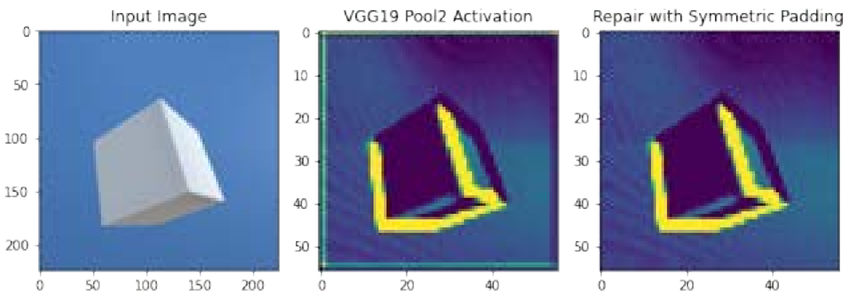

FIGURE B.3: The input image, VGG19's feature map activation with border effect artifacts and repaired with symmetric padding.

This solution works well in this particular dataset, as well as MNIST, as objects are not overlapping the borders and the background is of a single color without disturbances. For real-world images, a different solution would need to be applied. For example, a segmentation algorithm could differentiate between objects and the background, such that an analogous clear separation as in our dataset is available.

### B.1.3 Dataset Generation: Geometric Shapes

As a training dataset, we decided to start with easily detectable objects, based upon the geon idea by Biederman [40]. We built a piece of software using Unity[6], a game engine and development software often used for video game development and virtual reality applications. The code base for the dataset generation can also be found on GitHub. The images are generated to fit VGG19 at 224 by 224 pixels.

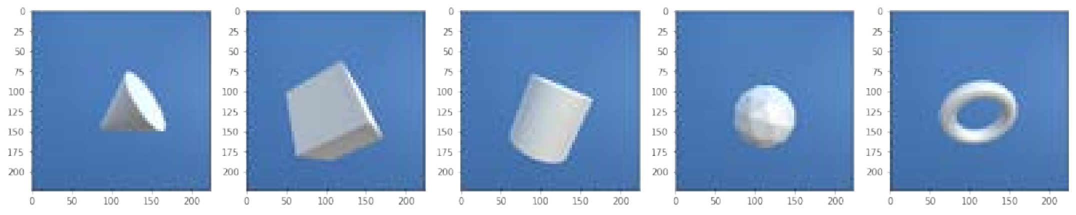

FIGURE B.4: Sample images of the geometric shapes dataset.

## B.2 Ablation Studies & Integration Tests

A number of ablation studies and integration tests have been conducted empirically and that are available in the form of Python notebooks. We refer the interested reader to our GitHub repository described in detail in Section B.1 of the Appendix. These experiments helped to get an intuition into the inner workings of the LCL and confirm that theoretically expected behavior is also seen in practice.

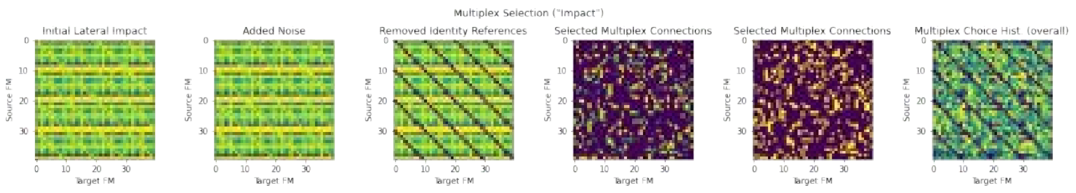

FIGURE B.5: Example of the LCL debug plot. From left to right, the plots show the lateral impact of every source feature map (y-axis) to every target feature map (x-axis) and how the stages of multiplex selection narrow down which cells are eventually activated. The right most plot shows a heat map of selections across all time steps.

The following capabilities and tests were conducted:

**Storing and Retrieving Patterns**[7]: The LCL represents its lateral connections through the learned kernel $K$. If the Hebbian learning mechanism during the forward pass is successful, it should be able to store multiple patterns and stabilize

[6]`https://unity.com/`

[7]For more details, see the Python notebooks `Ablation__LCL__SavingASinglePattern.ipynb` and `Ablation__LCL__SavingASinglePattern_MultiplexTargetSelection.ipynb`

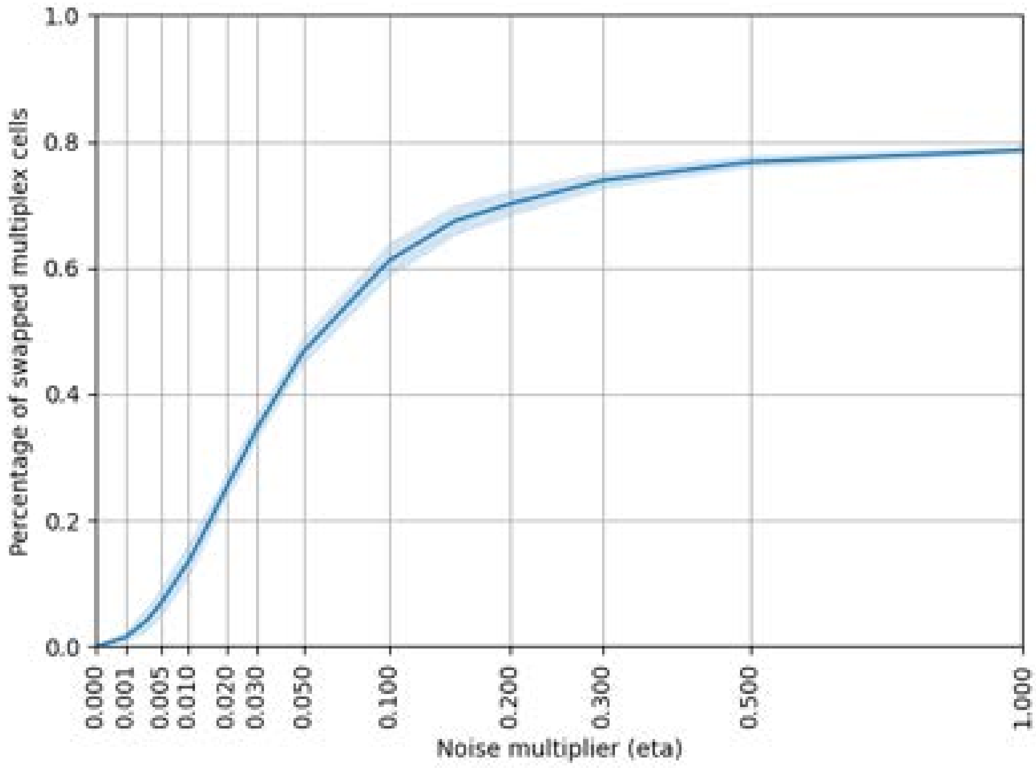

FIGURE B.6: Percentage of multiplex cells that are selected differently (y-axis) given the noise multiplier $\eta$ (x-axis).

them, given the same input patterns. To do this, we extracted the LCL code and fed a dataset containing either horizontal or vertical lines through the network. Over time, the LCL started to reproduce both types of lines successfully when excited with the corresponding input signals, but the output images exhibit a low-pass filter or blurring effect due to the size of the lateral kernels.

**Multiplex Cell Selection**[8]: The selection process of multiplex cells is one of the main driving forces of the LCL architecture. In this notebook we devised a set of debug plots to visualize the process and measure how the selections vary across time (for example if all cells are used or if a small subset dominates the selected cells). An example of such a plot is shown in Figure B.5.

**Lateral Kernel Initialization**[9]: There is a strong feedback loop acting between the kernel initialization, the magnitude of the noise during the multiplex selection process and dataset. We experimented with different multiplex selection processes and how they interact with various kernel initializations.

**Noise**[10]: Given that noise plays an important role, we set out to check the impact of different levels of noise. We concluded that the noise levels only change the multiplex selection up to a certain degree, but that an equivalence between the chosen value of $\eta$ and the percentage of multiplex cells that are chosen differently can be estimated empirically. Figure B.6 shows the percentage of changed multiplex cells given a TinyLateralNet with $n = 5$ and $d = 0$. In that

[8] For more details, see the Python notebook `Ablation__LCL__MultiplexSelection.ipynb`

[9] For more details, see the Python notebooks starting with `Ablation__LCL__KernelInitialization`

[10] For more details, see the Python notebook `TinyLateralNetwork_DebugMalsburgNoise.ipynb`

case, the TLN with $d = 0$ from our experiments in Chapter 4 switched around 50% of multiplex cells during the time that noise was applied.

## B.3 Experiment Figures

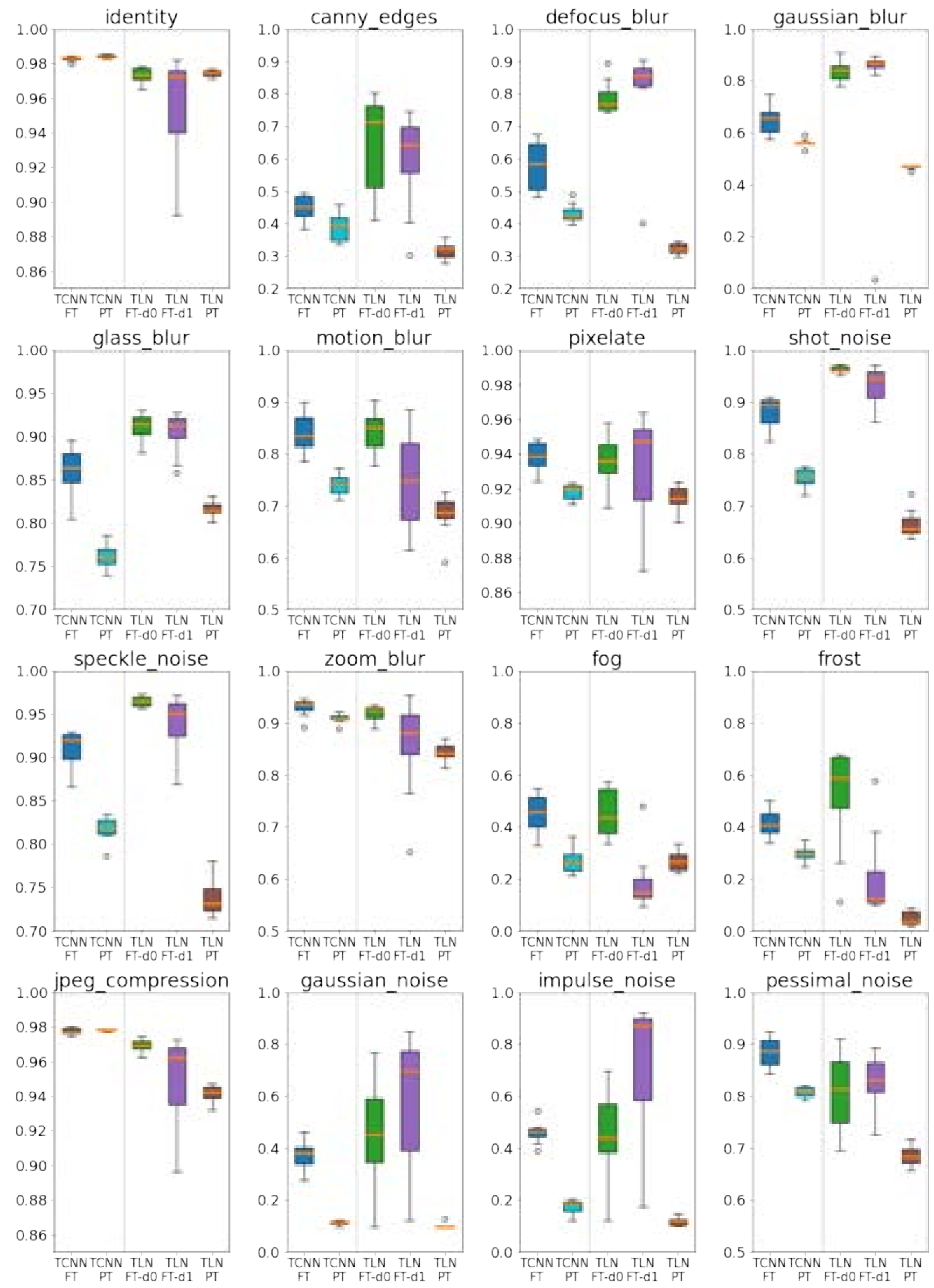

FIGURE B.7: Visualization of the results shown in Table 4.3 (Part 1). Abbreviations used are TinyCNN (TCNN), TinyLateralNet (TLN), Fully trained (FT) and Pre-trained (PT).

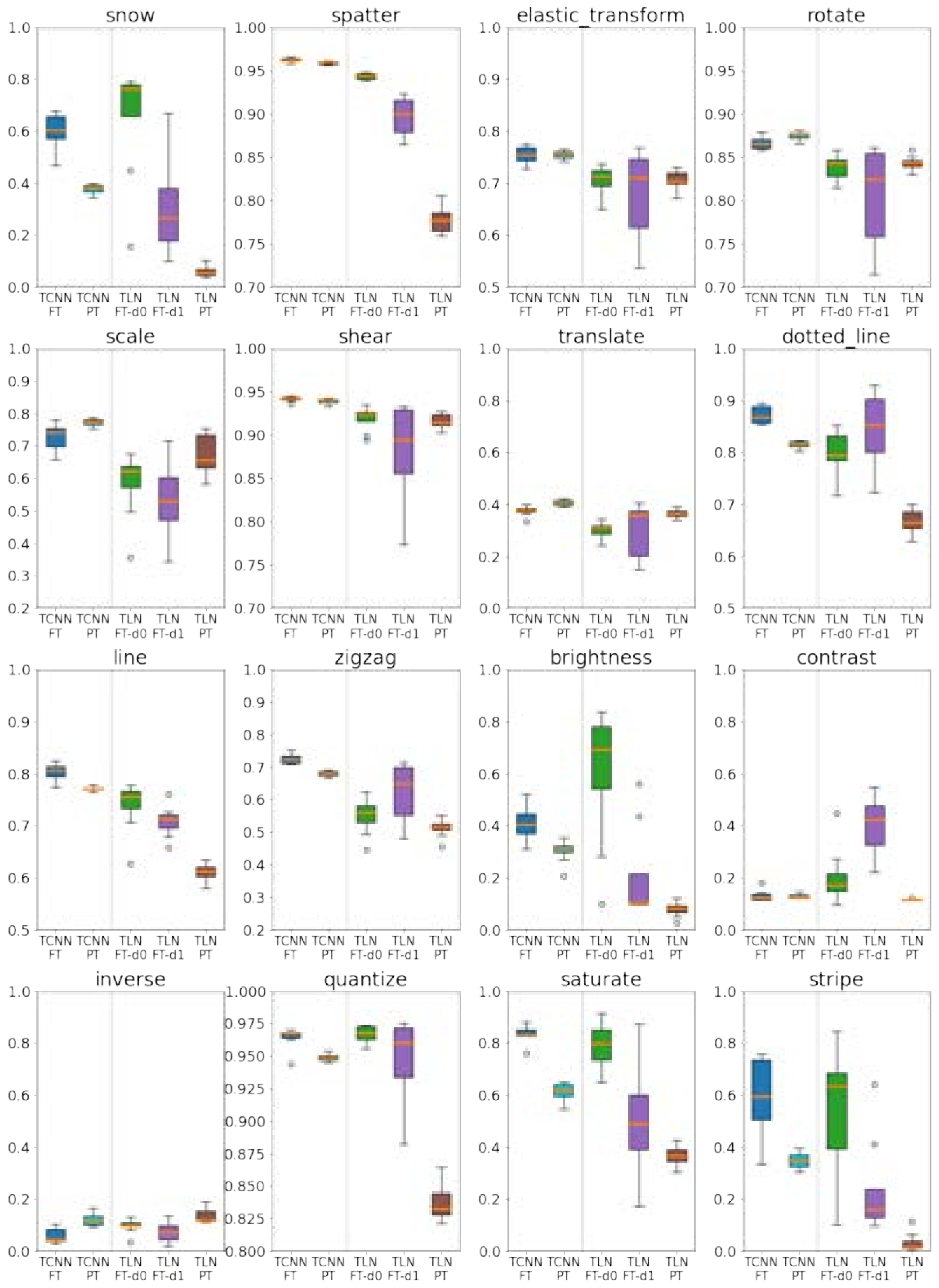

FIGURE B.8: Visualization of the results shown in Table 4.3 (Part 2). Abbreviations used are TinyCNN (TCNN), TinyLateralNet (TLN), Fully trained (FT) and Pre-trained (PT).

# Bibliography

[1] Alan M Turing. 'Computing Machinery and Intelligence'. In: *Parsing the turing test*. Springer, 2009, pp. 23–65.

[2] John Shalf. 'The Future of Computing Beyond Moore's Law'. In: *Philosophical Transactions of the Royal Society A* 378.2166 (2020), p. 20190061.

[3] David Patterson et al. 'Carbon Emissions and Large Neural Network Training'. In: *arXiv preprint arXiv:2104.10350* (2021).

[4] Alexey Dosovitskiy et al. 'An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale'. In: *arXiv preprint arXiv:2010.11929* (2020).

[5] Tom Brown et al. 'Language Models are Few-Shot Learners'. In: *Advances in neural information processing systems* 33 (2020), pp. 1877–1901.

[6] Julian Schrittwieser et al. 'Mastering Atari, Go, Chess and Shogi by Planning with a Learned Model'. In: *Nature* 588.7839 (2020), pp. 604–609.

[7] Alec Radford et al. 'Language Models are Unsupervised Multitask Learners'. In: *OpenAI blog* 1.8 (2019), p. 9.

[8] Alex Radford et al. *Better Language Models and Their Implications*. 2019. URL: `https://openai.com/blog/better-language-models/` (visited on 10/07/2022).

[9] Chitwan Saharia et al. 'Photorealistic Text-to-Image Diffusion Models with Deep Language Understanding'. In: *arXiv preprint arXiv:2205.11487* (2022).

[10] Google Research Brain Team. *Imagen*. 2022. URL: `https://imagen.research.google/` (visited on 10/07/2022).

[11] Will Douglas Heaven. *This horse-riding astronaut is a milestone in AI's journey to make sense of the world*. 2022. URL: `https://www.technologyreview.com/2022/04/06/1049061/dalle-openai-gpt3-ai-agi-multimodal-image-generation/` (visited on 10/07/2022).

[12] Gary Marcus. *What does it mean when an AI fails? A Reply to SlateStarCodex's riff on Gary Marcus*. 2022. URL: `https://garymarcus.substack.com/p/what-does-it-mean-when-an-ai-fails` (visited on 10/07/2022).

[13] David E Rumelhart, Geoffrey E Hinton and Ronald J Williams. 'Learning Representations by Back-Propagating Errors'. In: *nature* 323.6088 (1986), pp. 533–536.

[14] Warren S McCulloch and Walter Pitts. 'A Logical Calculus of the Ideas Immanent in Nervous Activity'. In: *The bulletin of mathematical biophysics* 5.4 (1943), pp. 115–133.

[15] Frank Rosenblatt. 'The perceptron: A Probabilistic Model for Information Storage and Organization in the Brain'. In: *Psychological review* 65.6 (1958), p. 386.

[16] Andrinandrasana D Rasamoelina, Fouzia Adjailia and Peter Sinčák. 'A Review of Activation Function for Artificial Neural Network'. In: *2020 IEEE 18th World Symposium on Applied Machine Intelligence and Informatics (SAMI)*. IEEE. 2020, pp. 281–286.

[17] Matthew D Zeiler and Rob Fergus. 'Visualizing and Understanding Convolutional Networks'. In: *European conference on computer vision*. Springer. 2014, pp. 818–833.

[18] David H Hubel and Torsten N Wiesel. 'Receptive Fields of Single Neurones in the Cat's Striate Cortex'. In: *The Journal of physiology* 148.3 (1959), p. 574.

[19] David H Hubel and Torsten N Wiesel. 'Receptive Fields, Binocular Interaction and Functional Architecture in the Cat's Visual Cortex'. In: *The Journal of physiology* 160.1 (1962), p. 106.

[20] David H Hubel and Torsten N Wiesel. 'Receptive Fields and Functional Architecture of Monkey Striate Cortex'. In: *The Journal of physiology* 195.1 (1968), pp. 215–243.

[21] Kunihiko Fukushima and Sei Miyake. 'Neocognitron: A Self-Organizing Neural Network Model for a Mechanism of Visual Pattern Recognition'. In: *Competition and cooperation in neural nets*. Springer, 1982, pp. 267–285.

[22] Yann LeCun et al. 'Backpropagation Applied to Handwritten Zip Code Recognition'. In: *Neural computation* 1.4 (1989), pp. 541–551.

[23] Louis Lapique. 'Recherches quantitatives sur l'excitation electrique des nerfs traitee comme une polarization.' In: *Journal of Physiology and Patholoolgy* 9 (1907), pp. 620–635.

[24] Donald O Hebb. *The Organization of Behavior: A Neuropsychological Theory*. Psychology Press, 2005.

[25] John J Hopfield. 'Neural Networks and Physical Systems with Emergent Collective Computational Abilities'. In: *Proceedings of the national academy of sciences* 79.8 (1982), pp. 2554–2558.

[26] Shay Gueron, Simon A Levin and Daniel I Rubenstein. 'The Dynamics of Herds: From Individuals to Aggregations'. In: *Journal of Theoretical Biology* 182.1 (1996), pp. 85–98.

[27] W Roepke. *Beobachtungen an Indischen Honigbienen insbesondere an Apis Dorsata F.* Tech. rep. Veenman, 1930.

[28] Andrea Crisanti, Daniel J Amit and Hanoch Gutfreund. 'Saturation Level of the Hopfield Model for Neural Network'. In: *EPL (Europhysics Letters)* 2.4 (1986), p. 337.

[29] Claude Berrou and Vincent Gripon. 'Coded Hopfield Networks'. In: *2010 6th International Symposium on Turbo Codes & Iterative Information Processing*. IEEE. 2010, pp. 1–5.

[30] John J Hopfield. 'Neurons With Graded Response Have Collective Computational Properties Like Those of Two-State Neurons'. In: *Proceedings of the national academy of sciences* 81.10 (1984), pp. 3088–3092.

[31] Adriano Barra, Matteo Beccaria and Alberto Fachechi. 'A New Mechanical Approach to Handle Generalized Hopfield Neural Networks'. In: *Neural Networks* 106 (2018), pp. 205–222.

[32] Dmitry Krotov and John J Hopfield. 'Dense Associative Memory for Pattern Recognition'. In: *Advances in neural information processing systems* 29 (2016).

[33] Mete Demircigil et al. 'On a Model of Associative Memory with Huge Storage Capacity'. In: *Journal of Statistical Physics* 168.2 (2017), pp. 288–299.

[34] Hubert Ramsauer et al. 'Hopfield Networks is All You Need'. In: *arXiv preprint arXiv:2008.02217* (2020).

[35] Ashish Vaswani et al. 'Attention is All you Need'. In: *Advances in Neural Information Processing Systems*. Ed. by I. Guyon et al. Vol. 30. Curran Associates, Inc., 2017. URL: `https://proceedings.neurips.cc/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf`.

[36] Dmitry Krotov and John Hopfield. 'Large Associative Memory Problem in Neurobiology and Machine Learning'. In: *arXiv preprint arXiv:2008.06996* (2020).

[37] Jeremy Freeman and Eero P Simoncelli. 'Metamers of the Ventral Stream'. In: *Nature neuroscience* 14.9 (2011), pp. 1195–1201.

[38] Ricardo Gattass, Charles G Gross and Julie H Sandell. 'Visual Topography of V2 in the Macaque'. In: *Journal of Comparative Neurology* 201.4 (1981), pp. 519–539.

[39] Serge O Dumoulin and Brian A Wandell. 'Population Receptive Field Estimates in Human Visual Cortex'. In: *Neuroimage* 39.2 (2008), pp. 647–660.

[40] Irving Biederman. 'Recognition-by-Components: A Theory of Human Image Understanding'. In: *Psychological review* 94.2 (1987), p. 115.

[41] Leon Gatys, Alexander S Ecker and Matthias Bethge. 'Texture Synthesis Using Convolutional Neural Networks'. In: *Advances in neural information processing systems* 28 (2015), pp. 262–270.

[42] Karen Simonyan and Andrew Zisserman. *Very Deep Convolutional Networks for Large-Scale Image Recognition*. 2015. arXiv: `1409.1556 [cs.CV]`.

[43] Bela Julesz. 'Visual Pattern Discrimination'. In: *IRE transactions on Information Theory* 8.2 (1962), pp. 84–92.

[44] Guo-qiang Bi and Mu-ming Poo. 'Synaptic Modifications in Cultured Hippocampal Neurons: Dependence on Spike Timing, Synaptic Strength, and Postsynaptic Cell Type'. In: *Journal of neuroscience* 18.24 (1998), pp. 10464–10472.

[45] Xiang Cheng et al. 'LISNN: Improving Spiking Neural Networks with Lateral Interactions for Robust Object Recognition'. In: *IJCAI*. 2020, pp. 1519–1525.

[46] Peter U Diehl and Matthew Cook. 'Unsupervised Learning of Digit Recognition Using Spike-Timing-Dependent Plasticity'. In: *Frontiers in computational neuroscience* 9 (2015), p. 99.

[47] Ravi Kothari and Kwabena Agyepong. 'On Lateral Connections in Feed-Forward Neural Networks'. In: *Proceedings of International Conference on Neural Networks (ICNN'96)*. Vol. 1. IEEE. 1996, pp. 13–18.

[48] Scott Fahlman and Christian Lebiere. 'The Cascade-Correlation Learning Architecture'. In: *Advances in neural information processing systems* 2 (1989).

[49] David J Montana, Lawrence Davis et al. 'Training Feedforward Neural Networks Using Genetic Algorithms'. In: *IJCAI*. Vol. 89. 1989, pp. 762–767.

[50] Bernd Porr and Paul Miller. 'Forward Propagation Closed Loop Learning'. In: *Adaptive Behavior* 28.3 (2020), pp. 181–194.

[51] Adam Kohan, Edward A Rietman and Hava T Siegelmann. 'Forward Signal Propagation Learning'. In: *arXiv preprint arXiv:2204.01723* (2022).

[52] Christoph von der Malsburg. 'Concerning the Neural Code'. In: *arXiv preprint arXiv:1811.01199* (2018).

[53] Christoph von der Malsburg, Thilo Stadelmann and Benjamin F Grewe. 'A Theory of Natural Intelligence'. In: *arXiv preprint arXiv:2205.00002* (2022).

[54] Norman Mu and Justin Gilmer. 'MNIST-C: A Robustness Benchmark for Computer Visionn'. In: *arXiv preprint arXiv:1906.02337* (2019).

[55] Chen-Yu Lee, Patrick W Gallagher and Zhuowen Tu. 'Generalizing Pooling Functions in Convolutional Neural Networks: Mixed, Gated, and Tree'. In: *Artificial intelligence and statistics*. PMLR. 2016, pp. 464–472.

[56] Gregory Cohen et al. 'EMNIST: Extending MNIST to Handwritten Letters'. In: *2017 international joint conference on neural networks (IJCNN)*. IEEE. 2017, pp. 2921–2926.

[57] Bobby Allyn. *The Google engineer who sees company's AI as 'sentient' thinks a chatbot has a soul*. 2022. URL: `https://www.npr.org/2022/06/16/1105552435/google-ai-sentient?t=1657451274053` (visited on 10/07/2022).

[58] Steve LeVine. *Artificial intelligence pioneer says we need to start over*. 2017. URL: `https://www.axios.com/2017/12/15/artificial-intelligence-pioneer-says-we-need-to-start-over-1513305524` (visited on 10/07/2022).

[59] Yann LeCun. *A Path Towards Autonomous Machine Intelligence*. 2022. URL: `https://openreview.net/pdf?id=BZ5a1r-kVsf` (visited on 10/07/2022).

[60] Gary Marcus. *Is "Deep Learning" a Revolution in Artificial Intelligence?* 2012. URL: `https://www.newyorker.com/news/news-desk/is-deep-learning-a-revolution-in-artificial-intelligence` (visited on 10/07/2022).

[61] Bilal Alsallakh et al. *Mind the Pad – CNNs can Develop Blind Spots*. 2020. arXiv: `2010.02178 [cs.CV]`.