Department of Information Engineering and Computer Science

Master's Degree in
Artificial Intelligence Systems

Final Dissertation

Backpropagation meets contrastive neuron-centric hebbian learning

Supervisor

Giovanni Iacca

Student

Riccardo Lunelli

Co-Supervisors

Cunegatti Elia, Ferigo Andrea

Academic year 2023/2024

Acknowledgements

Questo è forse uno dei capitoli più difficili da scrivere per me... Con questa tesi si conclude questo splendido percorso, pieno di emozioni ed esperienze che vanno oltre alle tantissime ed interessantissime cose imparate.

Vorrei quindi ringraziare in primis l'Università degli Studi di Trento per avermi dato la possibilità di seguire questo corso e soprattutto per avermi dato l'opportunità di fare il periodo di Erasmus in Norvegia, uno dei periodi più belli della mia vita.

Vorrei ringraziare in particolare il professor Giovanni Iacca che mi ha seguito in questa tesi e che mi ha anche aiutato nello scrivere un articolo per EvoStar2024.

Un altro ringraziamento molto sentito lo devo a Dimension ed in particolare a Diego e Daniele, che mi hanno dato l'opportunità di poter continuare a lavorare durante tutto il periodo degli studi con molta flessibilità ed autonomia. Lavorare e studiare assieme non è sempre stato facile ma non mi avete messo nessun paletto nel trovare il mio giusto equilibrio e credo che questa sia una cosa molto rara nel mondo del lavoro.

Non posso non ringraziare anche tutta la mia famiglia: Giorgia, Fabio, Lara e Martin che mi sono sempre stati vicini e che mi hanno aiutato, anche nella semplicità di avere un pasto pronto e in tante altre piccole cose, di concentrare meglio il mio tempo e le mie energie per completare questo percorso.

Un altro ringraziamenteo va a Erica, con cui ho passato alcuni dei momenti piu belli della mia vita. Tutte le notti in Norvegia passate ad aspettare l'aurora boreale che giocava a nascondino. Fino a trovarla sperduta nel bel mezzo del circolo polare artico da una casetta in mezzo al nulla piu assoluto ad osservarla stupetatti, senza parole per descriverla ma solamente emozioni per viverla. E grazie per tutti i momenti difficili in cui nonostante tutto hai scelto di starmi accanto.

Voglio anche ringraziare il mio gatto Bert: sei riuscito a rendermi più leggeri tanti giorni troppo indaffarati, facendomi sorridere con la tua stupidità e il tuo carattere forte. Te ne sei andato però troppo presto e ogni volta guardo la strada davanti casa con la speranza di vederti arrivare anche se so che non succederà. Mi manchi tanto e una parte di questo lavoro la dedico anche a te.

E poi ringrazio anche tutte le persone conosciute in questa esperienza, dai compagni di corso a tutti quelli che ho conosciuto in erasmus: grazie per i bei momenti e tutte le esperienze fatte assieme

Infine ringrazio tutti i miei amici che, in un modo o nell'altro mi hanno sostenuto e mi hanno reso leggeri e divertenti tanti momenti.

Contents

Abstract

Hebbian Learning, inspired by synaptic plasticity in biological neural networks, holds the potential to revolutionize deep learning by offering biological realism and enabling more efficient and continual learning capabilities. However, traditional Hebbian rules have struggled to scale to deeper architectures while maintaining a simple and readily adaptable training process. This thesis aims not to achieve state-of-the-art performance but to pave the way for a new approach to Hebbian Learning by introducing a more robust and extensible framework applicable to arbitrary network architectures.

The core contribution of this work is the Contrastive Hebbian Learning framework, which combines neuron-centric learning rates, contrastive learning principles, and the power of backpropagation. This method introduces a novel contrastive learning rule operating at the layer level, promoting the emergence of discriminative representations by encouraging separation between outputs of different classes while maintaining similarity between outputs of the same class. Importantly, it adopts a neuron-centric approach, significantly reducing the number of trainable parameters by assigning learning rates to individual neurons instead of synapses. Backpropagation is employed to optimize these neuron-centric learning rates, avoiding the need for computationally demanding evolutionary algorithms commonly used in Hebbian Learning.

Experimental results on MNIST, CIFAR-10, and STL-10 datasets demonstrate the effectiveness of the Contrastive Hebbian method, showcasing its ability to learn in deep linear networks and achieve competitive performance on these image classification benchmarks. While its performance on convolutional networks currently lags behind specialized methods like SoftHebb, the Contrastive Hebbian method distinguishes itself through its greater robustness to variations in width scaling and choice of activation function. This robustness simplifies hyperparameter tuning and increases the framework's adaptability to different network configurations. Furthermore, the method exhibits promising resistance to catastrophic forgetting in continual learning scenarios, particularly when applied to convolutional networks, as observed in experiments on the Split-MNIST task.

This thesis marks a significant step forward in the development of a more accessible and versatile Hebbian Learning framework for deep neural networks. In particular allowing to use such framework using arbitrary network architectures and being robust to the choice of hyperparameters. The Contrastive Hebbian method, with its simplicity, end-to-end training procedure, and inherent potential for continual learning and other dynamic tasks, provides a solid foundation for future research in biologically plausible deep learning. The work opens up exciting new directions, including exploring unsupervised extensions, devising strategies for complete removal of backpropagation, and investigating applications to a broader range of tasks beyond image classification.

1 Introduction

In the field of machine learning, back-propagation [41] is the most widely used algorithm for optimizing deep neural networks. Developed in the 1980s, back-propagation revolutionized the training of artificial neural networks, enabling the significant advancements we see today. Despite its success, back-propagation is not without its drawbacks, particularly its lack of biological plausibility and the requirement for supervised learning. Of course, other, more biologically plausible, approaches have been explored and it is still an active area of research. It is the case of Hebbian Learning [13], a macro group of rules that aims to mimic the behavior of the brain in artificial neural networks. This thesis explores an alternative approach: using back-propagation to learn the parameters of Hebbian Learning rules, aiming to combine the strengths of both methods.

1.1 Backpropagation

Backpropagation is a supervised learning algorithm that iteratively adjusts the weights of a neural network to minimize the error between the predicted and actual outputs. The process involves two phases: forward propagation and backward propagation. In forward propagation, input data passes through the network to generate an output. In backward propagation, the gradient with respect to the given loss function is calculated and propagated backward through the network, updating the weights with the corresponding partial derivative.

Effective Error Minimization

One of the most significant advantages of backpropagation is its ability to minimize a global error function effectively. By calculating the gradient of the error function with respect to the network weights, backpropagation provides a systematic way to adjust these weights to reduce the overall error function. This gradient descent approach ensures that the network gradually improves its performance on the training data, leading to more accurate predictions.

Versatility

Another advantage of backpropagation is its versatility. It can be applied to a wide range of neural network architectures such as feedforward networks, convolutional neural networks, and recurrent neural networks. Whether the task involves classification, regression, or sequence prediction, backpropagation provides a robust framework for training neural networks to perform at high levels of accuracy.

Highly optimized hardware

Thanks to the very important success of backpropagation, companies have developed specialized hardware and libraries to handle the huge computational demands of training neural networks. This includes NVIDIA GPUs, Google TPUs, and the new Apple ARM chips. These advancements in hardware enable backpropagation to run much faster and more efficiently, facilitating the training of very large networks. With this investment in specialized hardware and its optimization properties, backpropagation has become the most convenient and effective way to train deep neural networks.

Computational Cost

Backpropagation is very computationally intensive, requiring significant processing power and memory, especially for big networks. The need for extensive gradient calculations makes backpropagation very resource-intensive. Considering the fact that the most used optimizers use two or more times the

number of network parameters, for example, to store the momentum of the gradient. Training deep networks with backpropagation often requires specialized hardware, such as GPUs, and a large amount of memory. This computational demand can be a problem for those who have limited access to such hardware. Table 1.1 presents an overview with respect to the increase of the peak memory requirements of some of the most used optimizers.

Optimizer	**Memory Usage**
SGD	2× (params + grads)
SGD with Momentum [37]	3× (params + grads + momentum)
AdaGrad [8]	3× (params + grads + accumulators)
Adam [19]	4× (params + grads + momentum + velocity)
AMSGrad [40]	5× (params + grads + momentum + velocity + max velocity)

Table 1.1: Memory usage of different optimizers.

Biological Implausibility

One of the primary criticisms of backpropagation is its lack of biological plausibility [7]. The algorithm relies on the backward propagation of error signals through the network, a process that is not observed in biological neural systems. In the brain, learning occurs through more localized and distributed mechanisms, such as synaptic plasticity governed by more efficient and resource-demanding Hebbian principles.

1.1.1 Intense data demand

Moreover, backpropagation typically requires large collections of labeled data used in a supervised training setting. In many real-world scenarios, labeled data is scarce, expensive, and/or time-consuming to obtain. Conversely, biologically plausible networks are unsupervised by design and should solve this problem by miming the learning process happening in our brains: to learn new tasks or new concepts, the brain, usually needs only a few sets of samples.

Catastrophic forgetting

Catastrophic forgetting [28], is an important problem when using backpropagation: a neural network trained sequentially on multiple tasks forgets previously learned information upon learning new tasks. This occurs because the weight adjustments required for new tasks overwrite the weights optimized for previous tasks, leading to performance degradation on earlier tasks. This phenomenon poses a big challenge for developing models capable of continuous learning and adaptation.

1.2 Hebbian Learning

Hebbian Learning is based on the principle discovered by psychologist Donald Hebb in 1949, often summarized as "cells that fire together, wire together". This principle suggests that the synaptic strength between two neurons increases when they are activated simultaneously. It is a local learning rule, meaning that the synaptic adjustments depend only on the activities of the neurons connected by the synapse. This local aspect makes Hebbian Learning biologically plausible, as it mirrors the decentralized nature of neural processes occurring in the brain. There are several rules derived from Hebb's principle, all aiming to replicate the behavior of brain cells. Hebbian Learning closely mimics the synaptic plasticity observed in biological neural networks, making it a suitable model for understanding learning and memory in the brain. This biological realism, in theory, is crucial for enabling all the advantages of biological networks such as our brain.

Local Learning Rule

The adjustment of synaptic weights depends only on local information (the activations of the connected neurons). This localized nature aligns well with the decentralized structure of biological neural networks, where learning occurs at individual synapses without the need for global information. This can simplify the learning mechanisms and reduce the complexity of implementing and training neural networks.

Unsupervised Learning

Hebbian Learning typically operates without the need for labeled data or external error signals, which is an advantage in scenarios where such data is unavailable or difficult to obtain. This capability makes it suitable for environments where learning must occur autonomously, such as in certain robotic and sensory processing applications. However, some labeled data samples are required to optimize the Hebbian parameters and to assess the performance of models using Hebbian rules.

Lower computational demand

The Hebbian Learning rule is straightforward to implement, with synaptic weight changes based directly on the correlation between pre and post-synaptic neuron activations. In this way, each layer does not depend on subsequent layers when it comes to updating the weights and thus avoids the high amount of calculus needed by the backward pass of backpropagation. This can lead to more efficient algorithms and easier integration into various neural network models.

Continual Learning

Continual learning, allows a neural network to learn new tasks while retaining knowledge from previous tasks. Unlike backpropagation, which can lead to catastrophic forgetting, continual learning uses mechanisms to maintain long-term memory. Synaptic plasticity, naturally occurring with Hebbian Learning rules, allows for long-term potentiation (LTP) and long-term depression (LTD) [13]. LTP strengthens neural connections, making it easier to remember learned tasks, while LTD weakens less important connections, preventing saturation and allowing new learning.

A critique on Hebbian Learning

Although Hebbian Learning is a foundational concept in neuroscience and offers biological plausibility, it is not widely used in practical applications of machine learning. One significant reason is that when dealing with deep neural networks, the Hebbian Learning rules proposed in the literature struggle to make the network converge. Most implementations of Hebbian Learning are confined to shallow networks, which limits their applicability to more sophisticated tasks that require deeper architectures. Moreover, Hebbian Learning comes with a set of hyperparameters that need to be tuned, requiring additional effort to find the right combination. Finding such a combination usually involves evolutionary algorithms, which demand even more resources and time compared to backpropagation. Additionally, the number of Hebbian parameters is a multiple of the total number of weights in the network, depending on the rule used, making hyperparameter tuning more resource-intensive. In conclusion, while Hebbian Learning provides valuable insights and biological relevance, its practical limitations, especially in the context of deep networks and resource demands, make it less favorable compared to more efficient algorithms like backpropagation.

1.3 Contrastive Learning

Contrastive learning is a powerful and mature paradigm that focuses on learning the similarities and differences between samples. This paradigm is widely used in a wide variety of tasks where self-supervised or unsupervised learning is needed. Even supervised settings may benefit from contrastive learning, for example, to pre-train a model, as done for the famous model CLIP [38]. However, when

it comes to contrastive rules used in biologically plausible networks some uncommon practices are employed as we will see in the next chapter.

1.4 Integrating Hebbian Learning and Backpropagation

As we have seen so far, both approaches have their own advantages and disadvantages, but backpropagation is still the most adopted algorithm in the majority of scenarios. The aim of this thesis is to make an important step in bridging the gap between Hebbian Learning and backpropagation. With this work, I will attempt to combine Hebbian Learning and backpropagation to determine if we can integrate the advantages of both while mitigating some of their limitations. Specifically, the focus will be on eliminating the uncommon and specific strategies commonly used in Hebbian Learning when applied to deep neural networks. This includes reducing the number of Hebbian parameters and using backpropagation to train the parameters of a novel Hebbian rule, proposed for the first time in this thesis. As a result, this new framework will be flexible and robust enough to adapt to different architectures and resilient against hyperparameter changes, rather than aiming to achieve the highest accuracy.

1.5 Thesis structure

This thesis is structured as follows: Chapter 2 provides an extensive literature review of related works on Hebbian Learning, Contrastive Learning, and the interplay between backpropagation and Hebbian plasticity. Chapter 3 outlines the motivations and key objectives driving this research. Chapter 4 delves into the methodology, presenting the Contrastive Hebbian Learning rule, its mathematical derivation, and the integration of neuron-centric updates and backpropagation for training. Chapter 5 establishes the experimental setup, detailing the datasets and tasks, the network architectures, and the baseline methods used for evaluation. Chapter 6 presents the experimental results, analyzing the performance of the Contrastive Hebbian method on various image classification and continual learning tasks. Finally, Chapter 7 summarizes the contributions, discusses the advantages and limitations of the proposed method, and outlines promising future research directions.

2 Related works

2.1 Hebbian Learning

Hebbian Learning, known for its principle "neurons that fire together wire together," is a family of rules that defines how synaptic weights should change based only on pre and post-synaptic activations. The most basic rule, first proposed by Donald Hebb [13], is:

$$\Delta w_{ij} = \eta_{ij} x_i y_j \tag{2.1}$$

where η_{ij} is a learning rate, x_i is the pre-synaptic activation, and y_j is the post-synaptic activation. This rule describes the update for a single synaptic weight between two neurons. However, this rule doesn't account for the unbounded growth of synaptic weights, so normalization techniques are usually employed.

2.1.1 Oja's rule

In this context the Oja's rule [35] addresses the normalization problem by adding a discount factor:

$$\Delta w_{ij} = \eta_{ij}(y_j x_i - y_j^2 w_{ij}) = \eta_{ij} y_j (x_i - y_j w_{ij}). \tag{2.2}$$

Here the variables are the same as before and w_{ij} represents the actual synaptic weight. By subtracting the weight multiplied by the square of the post-synaptic activation, this rule stabilizes the weight and prevents uncontrolled growth.

2.1.2 Anti-Hebbian rule

Other approaches contrasting the problem of unbounded growth are proposed as well, such the Anti-Hebbian rule [10]. It functions as the inverse of the Hebbian rule, decreasing synaptic strength when both pre and post-synaptic neurons are simultaneously active:

$$\Delta w_{ij} = -\eta_{ij} x_i y_j. \tag{2.3}$$

Anti-Hebbian Learning is believed to be crucial for homeostasis and stability in biological neural networks, preventing runaway excitation and maintaining balanced neural activity. It is also implicated in specific neural computations and adaptive behaviors, contributing to various learning mechanisms in the brain.

2.1.3 ABCD rule

One of the most used and well-working Hebbian rules is the so-called ABCD rule [42] that has shown interesting results in a lot of works.

$$\Delta w_{ij} = \eta_{ij}(A_i x_i + B_j y_j + C_{ij} x_i y_j + D_{ij}) \tag{2.4}$$

where A_i and B_j are coefficients assigned to the pre-synaptic and post-synaptic activations respectively, C_{ij} is the coefficient for the product of pre and post-synaptic activations, and D_{ij} is a bias term. Other rules can be derived from that rule, setting to zero any of the hyperparameters, it is the case of the ABC rule:

$$\Delta w_{ij} = \eta_{ij}(A_i x_i + B_j y_j + C_{ij} x_i y_j). \tag{2.5}$$

2.1.4 Hebbian Descent

Melchior and Wiskott [29] proposed a very interesting approach where a Hebbian Rule with the same updates as backpropagation is applied to single-layered networks. In particular, they show that for a positive derivative of the activation function, this rule is equivalent to backpropagation:

$$\Delta_{HD}\mathbf{W} = -\eta(\mathbf{x}-\mu)\frac{\partial\mathcal{L}(\mathbf{t},\mathbf{h})}{\partial\mathbf{h}}^T = -\eta(\mathbf{x}-\mu)\mathcal{E}(\mathbf{t},\mathbf{h})^T, \tag{2.6}$$

$$\Delta_{HD}\mathbf{b} = -\eta\frac{\partial\mathcal{L}(\mathbf{t},\mathbf{h})}{\partial\mathbf{h}} = -\eta\mathcal{E}(\mathbf{t},\mathbf{h})^T, \tag{2.7}$$

where $\frac{\partial\mathcal{L}(\mathbf{t},\mathbf{h})}{\partial\mathbf{h}}$ is the derivative of the loss function, which is equal to the error signal $\mathcal{E}(\mathbf{t},\mathbf{h})$; $\mathbf{t}$ are the target values in one-hot encoding, $\mathbf{x}$ is the input, and $\mathbf{h}$ is the output of the neurons, equal to $\phi((\mathbf{x}-\mu)\mathbf{W}+\mathbf{b})$; μ is the mean of the input activations. Considering the squared error loss $\mathcal{L}(\mathbf{t},\mathbf{h}) = \frac{1}{2}(\mathbf{h}-\mathbf{t})^T(\mathbf{h}-\mathbf{t})$, the corresponding error signal (its derivative) is $(\mathbf{h}-\mathbf{t})^T$. Thus, the Hebbian update is the following:

$$\begin{aligned}\Delta_{HD}\mathbf{W} &= -\eta(\mathbf{x}-\mu)\left(\phi((\mathbf{x}-\mu)\mathbf{W}+\mathbf{b})-\mathbf{t}\right)^T \\ &= \underbrace{\eta(\mathbf{x}-\mu)\mathbf{t}^T}_{\text{Sup.Hebb}} - \underbrace{\eta(\mathbf{x}-\mu)\mathbf{h}^T}_{\text{Unsp.Hebb}},\end{aligned} \tag{2.8}$$

$$\begin{aligned}\Delta_{HD}\mathbf{b} &= -\eta\left(\phi((\mathbf{x}-\mu)\mathbf{W}+\mathbf{b})-\mathbf{t}\right) \\ &= \underbrace{\eta\mathbf{t}}_{\text{Sup.Hebb}} - \underbrace{\eta\mathbf{h}}_{\text{Unsp.Hebb}}.\end{aligned} \tag{2.9}$$

To train neural networks in a classification task, the cross-entropy loss function is usually employed: $\mathcal{L}(\mathbf{t},\mathbf{h}) = -\mathbf{t}^T\ln(\mathbf{h}) - (1-\mathbf{t})^T\ln(1-\mathbf{h})$, and we know that the derivative of such loss with respect to the weights is, after some simplifications, $\frac{\partial\mathcal{L}(\mathbf{t},\mathbf{h})}{\partial\mathbf{w}} = \mathbf{x}(\mathbf{t}-\mathbf{h})^T$, which is equivalent to the Hebbian descent rule, considering the sample mean equals zero. This work is very important because it demonstrates the biological plausibility of using backpropagation only on the last layer of a neural network, where the backward pass can be simplified with a simple Hebbian rule.

2.1.5 Winner Take All strategy

Moving towards greater biological plausibility, the Winner-Take-All (WTA) [3] family of Hebbian rules has shown interesting improvements. The principle is straightforward and aligns well with biological observations: only the neuron with the highest activation within a specific group receives a synaptic weight increase. This competitive learning mechanism promotes specialization among neurons, allowing them to become selectively responsive to particular features or patterns in the input.

This strategy well aligns with some biological principles happening in the brain:

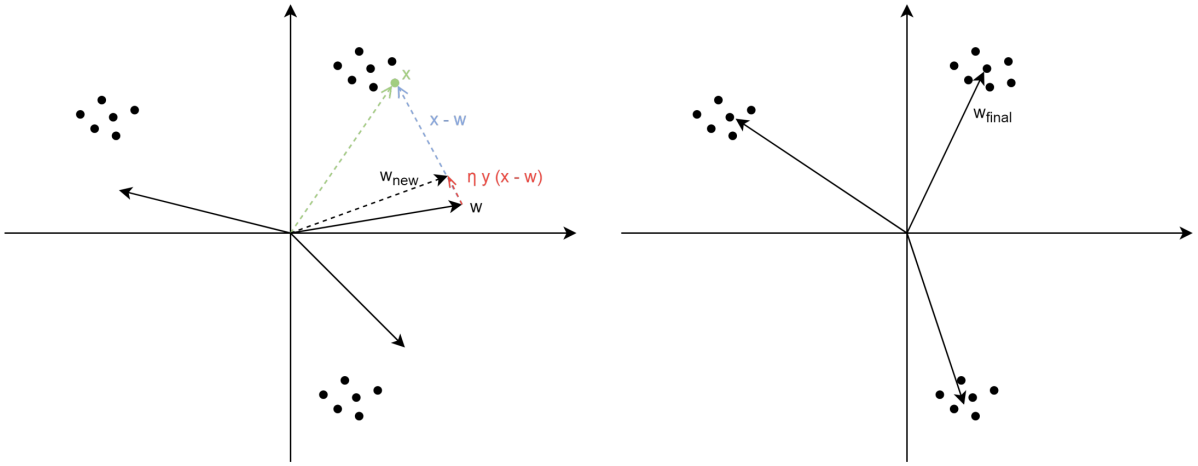
- *Resource competition*: In biological neural networks, neurons compete for limited resources such as neurotransmitters, energy, and connections. WTA reflects this competition by allowing only the most active neurons to strengthen their connections, effectively claiming the resources.

- *Lateral Inhibition*: WTA mechanisms can be implemented through lateral inhibition, a common pattern in biological neural circuits [4]. In this scheme, active neurons suppress the activity of their neighbors, implementing a competition where the strongest neuron inhibits others and emerges as the "winner".

- *Implicit Pruning*: During brain development, synaptic connections undergo extensive pruning, where unnecessary or weak connections are eliminated. WTA could contribute to this process by selectively strengthening the connections of the most active neurons, leading to having never-active neurons that can be considered pruned.

This approach works well when the input data is well separated, making it easier for such an algorithm to cluster the samples effectively. In practice, after training, each of the winning neurons can be seen as the centroid of the respective cluster of samples. While WTA Hebbian Learning is powerful, it also comes with challenges, particularly when dealing with overlapping clusters or noisy data. In such cases, the algorithm might struggle to form distinct clusters.

Figure 2.1: A graphical representation of how neurons converge in Hebbian WTA. The figure is taken from [23]



(a) Update step

(b) Final position after convergence

Hebbian Principal Component Analysis

As we have seen with Hebbian WTA, only one neuron activates when the corresponding pattern is presented. Neural networks trained with back-propagation exhibit more distributed representations, with more neurons active, encoding different properties of the input [22]. The rule proposed by Becker and Plumbley [2] aims to use a Hebbian rule that allows the weights to perform a PCA. This is done by minimizing the representation error defined as:

$$L(w_i) = E\left[\left(x - \sum_{j=1}^{i} y_j w_j\right)^2\right] \tag{2.10}$$

where i refers to the i^{th} neuron in the layer and $E[\cdot]$ is the mean operator. This loss leads to the following Hebbian update:

$$\Delta w_i = \eta y_i \left(x - \sum_{j=1}^{i} y_j w_j\right) \tag{2.11}$$

Lagani et al. [22] showed that using such a PCA-oriented approach leads to more promising results over WTA Hebbian Learning. However, they also noted that this approach is still not suitable for training deep networks and further research in the field is needed.

2.1.6 Soft WTA

A further and more promising extension of that paradigm is the Soft WTA approach, proposed by Moraitis et al. [34], which distributes the update among the most active neurons rather than only to the "winner". This is achieved through a softmax activation function, ensuring that the sum of all activations of postsynaptic neurons in a given layer equals 1:

$$y_k = \frac{b^{u_k}}{\sum_{l=1}^{K} b^{u_l}} = \frac{e^{\frac{u_k}{\tau}}}{\sum_{l=1}^{K} e^{\frac{u_l}{\tau}}}. \tag{2.12}$$

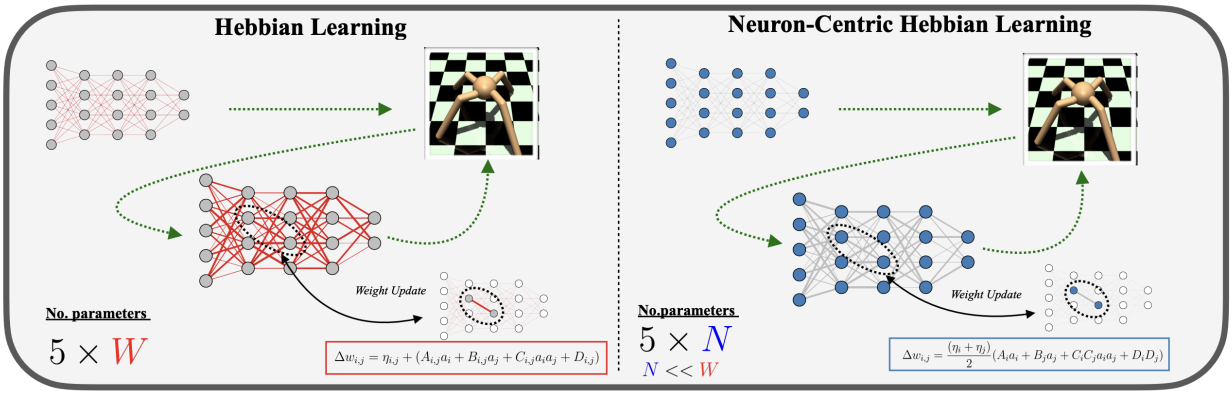
Here, u_k represents the weighted output of neuron k, $b^{uk} = e^{\frac{u_k}{\tau}}$, and τ is a temperature parameter. Smaller values of τ result in greater similarity to the WTA algorithm, while larger values distribute the update more smoothly among neurons.

$$\Delta w_{ik}^{(\text{SoftHebb})} = \eta \cdot y_k \cdot (x_i - u_k \cdot w_{ik}). \tag{2.13}$$

In their work, Moraitis et al. [34] achieved very promising results. They tested the algorithm in two settings, using both a single-layer and a two-layer network, each with 2000 neurons. They obtained impressive accuracies of (96.31 ± 0.06)% and (97.80 ± 0.02)% for their 1 and 2-layer networks, respectively. When applied to CIFAR-10, SoftHebb achieved an accuracy of 50.27%. Moraitis et al.demonstrate that this approach exhibits greater robustness to noise in the input data compared to backpropagation. Despite these promising results discussed in Section 2.1.6, the training method employed by Moraitis et al. misaligns from the standard workflow commonly used to train models. As stated in their paper: "Namely, for each neuron, we found the label of the training set that makes it win the WTA competition most often.". This means that after the training procedure, the neuron that mostly activates for a given layer is the one considered for providing the probability score of that label. This non-standard training procedure might lead to challenges for broader adoption and integration into typical machine learning pipelines where an end-to-end approach is needed.

2.1.7 Neuron-centric Hebbian Learning

Figure 2.2: A comparison of the vanilla Hebbian Learning and the Neuron-Centric Hebbian Learning. In particular, it is shown how Hebbian Learning uses a weight for each synapse and the Neuron-Centric version uses one parameter per neuron. The figure is taken from [9]



A novel and promising approach, proposed by Ferigo et al. [9], aims to significantly reduce the number of parameters. This method shifts the focus of Hebbian Learning from a synaptic-centric perspective to a neuron-centric perspective. Instead of assigning one value for the learning rate and the ABCD coefficients to each synapse, a single value for each coefficient is associated within each neuron. Since the number of neurons is typically orders of magnitude smaller than the number of synapses, this approach leads to a substantial reduction in hyperparameters.

In this work, the ABCD rule was taken as a reference, and the modified rule is the following:

$$\Delta w_{ij} = \frac{(\eta_i + \eta_j)}{2} (A_i x_i + B_j y_j + C_i C_j x_i y_j + D_i D_j) \tag{2.14}$$

where $\frac{(\eta_i + \eta_j)}{2}$ represents the average learning rate of neurons i and j, A_i and B_i are the coefficient for the pre and post-synaptic activations respectively, C_iC_j is the product of the C parameters of neuron i and j and the same for the bias term D_iD_j.

This neuron-centric approach can achieve a reduction of approximately 97% in hyperparameters. The authors evaluated this method in two simulated robotic locomotion tasks, demonstrating that performance remains competitive with the original synaptic-centric approach.

To further explore the potential of this framework, the authors propose a weightless version. Instead of storing synaptic weights, they are calculated dynamically at each time step using a history of recent pre- and post-synaptic activations. This history, which has a dimensionality equivalent to

the number of neurons, provides an approximation of the synaptic weights. While slightly less efficient and requiring more computation, this weightless approach offers a compelling advantage in scenarios where memory resources are highly limited.

2.1.8 Limitations

While Hebbian Learning provides a foundational framework for understanding neural plasticity, existing literature mainly focuses on shallow neural networks. This is a problem because it limits the ability of a network to find more complex patterns and thus it can be used only for very simple tasks.

Another problematic issue with this type of Hebbian rules is the vast number of hyperparameters that require to be optimized. For example, the standard Hebbian rule necessitates as many learning rates as there are weights in the network. The ABCD rule, while offering greater flexibility and power, demands the tuning of up to five times that number of hyperparameters. Evolutionary algorithms are often employed to find these parameters. However, when a differentiable loss function is available, employing such algorithms becomes inefficient compared to backpropagation. Backpropagation is much more effective and avoids the significant overhead associated with evolutionary algorithms. This makes Hebbian Learning well suited only for reinforcement learning scenarios where the reward signal is not differentiable and backpropagation cannot be used.

2.2 Hebbian plasticity and convolutions

This section explores the intersection of Hebbian plasticity and convolutional neural networks (CNNs), a powerful architecture for processing correlated data like images.

2.2.1 Biological plausibility

While CNNs are one of the most effective approaches at solving tasks involving correlated signals, their core mechanism of convolving a kernel over an image lacks direct biological plausibility. The computational efficiency of shifting kernels on GPUs doesn't translate well to neuronal circuits. However, by reframing the input image as a collection of patches, each considered as an individual sample, convolution can be interpreted as a more biologically plausible linear layer operation [36].

2.2.2 Some implementations

Several attempts have been made to train convolutional kernels using Hebbian Learning rules:

- Amato et al. [1] employed a WTA mechanism, achieving promising results on shallow networks. However, these methods struggled with convergence as network depth increased, confirming the limitations found for the linear layers.

- Lagani et al. [23] introduced a semi-supervised approach using Hebbian PCA. Their method outperformed backpropagation and Variational Autoencoders when labeled data was scarce (up to 5% of the dataset). However, performance did not improve with increasing labeled data or the addition of subsequent layers.

- Miconi [31] observed that Hebbian Learning tends to induce simple, Gabor-like features in early layers. These filters resemble edge detectors and are effective at capturing low-level features. However, as you go deeper into the network, you need to learn more complex, abstract representations. Gabor-like filters are not expressive enough to capture these higher-level features, leading to a bottleneck in the network's learning capacity as its depth increases.

2.2.3 State of the art

Journé et al. [18] proposed a very interesting framework addressing all the problems seen in all the other works. In particular, their approach has several improvements, first of all, it is extended to convolutions, it does not require any feedback propagation, time-locking updates, or other complex and non-common techniques to work. This method can achieve very good results: accuracies on MNIST, CIFAR-10, STL-10, and ImageNet, respectively are 99.4%, 80.3%, 76.2%, and 27.3%. The most interesting fact is that this is the first work that allows for deep networks to work with a Hebbian Learning setting. Their proposal involved the use of Anti-Soft-WTA where the winning neuron takes

the positive update while the other neurons get a negative update. In this way, they showed for the first time that increasing the number of channels at each layer leads to an increase in performance as the network becomes deeper. Another interesting approach proposed in this work is to have a dynamic learning rate decay, defined at neuron level, in order to decrease it as the weights of the neurons converge:

$$\eta_i = \eta \cdot (r_i - 1)^q \tag{2.15}$$

where q is a hyperparameter, set to 0.5, r_i is the actual norm of the neuron, and η is the initial learning rate, defined at the layer level. Note that convergence in this scenario is defined when a neuron reaches a norm of 1.

In this work, a custom and parametrizable activation function is used, namely the Triangle function introduced by Coates et al. [6]. It is a modification of the Rectified Polynomial Unit (RePU) [21], where the mean activation is subtracted before applying the RePU. The formulas of these activation functions are as follows:

$$RePU(u) = \begin{cases} u^p, & \text{for } u > 0 \\ 0, & \text{for } u \leq 0 \end{cases} \tag{2.16}$$

$$Triangle(u_j) = RePU(u_j - \bar{u}). \tag{2.17}$$

Note that in this case, p is a hyperparameter that makes this function parametrizable. In the paper, different values of p are employed for different layers. Adding hyperparameters at each layer makes this approach too specific and parametrizable, meaning less robustness to variation of such hyperparameters and more complexity in finding the correct set of parameters.

2.2.4 Limitations

Despite the impressive improvements of this work, to have such results an intensive grid search is employed: for every layer different initial learning rates are used, different temperatures for the softmax activation, different kernel sizes, pooling operators, and the p value of the Triangle activation function. This huge amount of manually tuned hyperparameters makes it very fragile when changing one of them may lead to a drastic decrease in performance Another limitation is that they trained only convolutional layers, with a linear head on the top, trained with backpropagation. These two aspects make this approach still incomplete, posing challenges when it comes to creating arbitrary deep network architecture [18]. But that's not all: it still misses an end-to-end procedure for training such networks. They used only one epoch to train the convolutional layers, stating that 5000 iterations are the optimal number of weight updates for this approach and the linear head is trained in a second step for 50 iterations with the Adam optimizer. Another consideration should be made because they only employ convolutional layers only when using Hebbian rules and a complete architecture with multiple linear layers after the convolutions is missing.

2.3 Contrastive Learning

Contrastive learning is a powerful learning paradigm that focuses on learning representations by distinguishing between similar and dissimilar examples. Instead of predicting explicit labels, contrastive learning encourages the model to group similar instances together while pushing dissimilar ones apart. This approach has become highly successful in self-supervised learning, where models learn from unlabeled data by creating their own positive and negative pairs. Contrastive learning has led to significant advances in image representation learning (e.g., SimCLR [5], MoCo [12]), and has also shown promise in natural language processing, reinforcement learning, and audio processing. The ability to learn from unlabeled data, improve generalization, and facilitate transfer learning has made contrastive learning an essential tool in modern machine learning.

Biologically plausibility is not only limited to Hebbian Learning but can be extended in other scenarios such as time-spiking neural networks (they will not be covered in this thesis) [27] and even in Contrastive Learning.

2.3.1 Forward forward

The forward-forward (FF) paradigm, introduced by Hinton [14], offers a novel approach to network training through contrastive learning. This method eliminates the backward pass, relying only on a forward pass, thus achieving the advantages discussed in the introduction for biologically plausible networks. As a contrastive learning approach, FF requires the creation of positive and negative samples. In this framework, labels are appended to the input data. Specifically, positive samples are augmented with a one-hot encoding vector representing the correct label, while negative samples receive a one-hot encoding corresponding to an incorrect label. The key innovation lies in maximizing the overall squared neural activations for positive samples and minimizing them for negative samples. This allows for training one layer at a time and to have a biologically plausible training framework.

$$G_{\text{pos}} = \sum_j y_{\text{pos},j}^2, \quad G_{\text{neg}} = \sum_j y_{\text{neg},j}^2 \tag{2.18}$$

$$L_{\text{pos}} = \log\left(1 + e^{\theta - G_{\text{pos}}}\right), \quad L_{\text{neg}} = \log\left(1 + e^{G_{\text{neg}} - \theta}\right). \tag{2.19}$$

Equation 2.19 presents the L_{pos} and L_{neg} losses that are minimized during training, θ is a hyperparameter, set to 2 in the original paper, to ensure that the squared activations are significantly above and below this threshold for positive and negative samples, respectively. To calculate weight updates, the derivative of the loss is computed. While for convenience, the `backward()` function can be used in frameworks like `PyTorch`, this does not compromise biological plausibility since the weight updates depend solely on pre and post-synaptic values.

This method achieves competitive performance on MNIST and CIFAR10, with an accuracy of 98.6% on MNIST and around 56% on CIFAR10.

Although this paradigm seems very promising, it requires the creation of positive and negative examples, making it not straightforward to implement. Moreover, at inference time, it requires more forward passes as the number of classes for the task increases. A sample to be classified must be fed to the network with all possible label encodings, and the one that has the highest squared activation is the one with the predicted label. This increases a lot the overhead and makes it not usable in a real-world scenario where speed of inference is a key element.

2.3.2 Forward-Forward and Hebbian Learning

While the Forward-Forward algorithm offers a biologically plausible alternative to backpropagation, its connection to established theories of neural learning has remained largely unexplored. A recent work by Terres-Escudero et al. [44] bridges this gap by demonstrating a direct equivalence between Forward-Forward and Hebbian Learning under specific conditions. In particular, they showed that using an Euclidean norm as the goodness function driving the local learning, the resulting Forward-Forward algorithm is equivalent to a neo-Hebbian Learning Rule [44]:

$$\frac{\partial w_{ij}}{\partial t} \doteq \Delta_t w_{ij} = M(t) g(y_j) x_i \tag{2.20}$$

where $M(t)g(y_j)x_i$ represent a *third factor* modulating the update at time t and $g(\cdot)$ is an aggregation operator for the spiking activity y_j of neuron j. Note that this method introduces spiking neural network principles [27], that are not covered in this thesis. They refer to this formulation as Hebbian FFA. The authors compare the performance of Hebbian FFA implemented in spiking neural networks to its analog counterpart on the MNIST dataset. To effectively use the spiking neural network properties they employed a batch size of 1. Results show that both versions achieve similar accuracy and exhibit comparable latent space properties, suggesting a strong link between Forward-Forward and biological learning mechanisms. This work highlights the potential of Forward-Forward to be implemented on neuromorphic hardware, leveraging the speed and energy advantages of these systems and thus its inherent biological plausibility.

2.3.3 Contrastive Hebbian Learning

Contrastive Hebbian Learning (CHL) is an established method initially used for training Boltzmann Machines [11]. The main idea behind CHL is to have feedback connections between layers, allowing information to flow back through the network, mimicking backpropagation while maintaining biological plausibility. This is achieved by employing two different phases: a "free" phase where the network operates normally, and a "clamped" phase where the output neurons are fixed at desired values, allowing the effects to propagate through the feedback connections. The update of the weights is based on the difference between the free and clamped states of the network.

Xie and Seung [46] demonstrate that under specific conditions, particularly in a multi-layer perceptron with linear output units and weak feedback connections, the change in network state due to clamping the output neurons in CHL is equivalent to the error signal propagated in backpropagation, differing only by a scalar factor. This equivalence suggests that backpropagation's functionality can be realized through a Hebbian-type learning algorithm, which may be more suitable for biological implementation. However, these kinds of algorithms are very sensible to numerical instability and not really usable in a real-world scenario. Another novel work by Høier and Zach [15] presents a new method called dual propagation, which attempts to bridge the performance gap to backpropagation without requiring special tricks for numerical instability. Despite its effectiveness, this method works well in an all-to-all connected network, while when it comes to a deep-layered network the overhead increases with the number of layers added. Moreover, this work is a theoretical overview only, without any test performed, not even on easy tasks such as MNIST.

2.4 Backpropagation and Hebbian plasticity

As seen so far, Hebbian plasticity demonstrates many theoretical advantages over backpropagation. However, none of the previous works have presented a fully functional, generalizable, simple, and well-performing approach to compete with backpropagation.

2.4.1 Learning to learn

A very interesting field of research has shifted the focus to combine both approaches to get the best from both worlds. Miconi [30], in his preliminary work, showed how backpropagation can be used to train both the weights and the Hebbian parameters. This novel approach unlocks new capabilities of the neural network, allowing it to "learn to learn" [30]. In particular, a time-dependent Hebbian plasticity is maintained by a Hebbian trace:

$$Hebb_k(t) = (1-\gamma) * Hebb_k(t-1) + \gamma * x_k(t) * y(t) \tag{2.21}$$

where $x_k(t)$ and $y(t)$ are the activities of the pre and post-synaptic neurons respectively, and γ is a time constant determining how much the trace is influenced by new data. Here, the output of a neuron takes into consideration the Hebbian trace for its calculation, and the formula is as follows:

$$y(t) = \tanh\left\{\sum_{k \in inputs} [w_k x_k(t) + \alpha_k Hebb_k(t) x_k(t)] + b\right\} \tag{2.22}$$

where α_k is the Hebbian Learning rate vector for the k^{th} neuron, previously indicated as η; b is the bias term, and w_k is the weight vector of the k^{th} neuron. During the backpropagation step, both the weights and the set of learning rates α are updated. The motivation is that, instead of using an evolutionary algorithm to find the Hebbian parameters, Miconi [30] allows the chain rule of back-propagation to find the optimal combination of such parameters.

This novel solution showed promising results in a variety of tasks, such as pattern completion, one-shot learning of arbitrary patterns, and reversal learning, where networks trained with backpropagation alone struggled to succeed.

2.4.2 Differentiable plasticity

In his following work "Differentiable Plasticity: Training Plastic Neural Networks with Backpropagation," Miconi et al. [32] extended this mechanism to Recurrent Neural Networks and demonstrated its

effectiveness on meta-learning tasks such as pattern memorization, the Omniglot task [24], and even in Reinforcement Learning scenarios. In all these tasks, the approach showed increased performance with respect to the use of vanilla back-propagation only.

2.4.3 Backpropamine

In the third work of this saga, [33] [33], introduces two new mechanisms:

- **Retroactive Neuro-modulation**: In biological brains, neuro-modulation is achieved via neurotransmitters such as dopamine, allowing the brain to decide whether or not to modify its connectivity. This enables the brain to filter out irrelevant states and focus only on important information. This neuro-modulation can be mimicked by a neuron that decides how much weight to give to the current state. Consequently, the formula is as follows:

$$Hebb_{ij}(t+1) = Clip(Hebb_{ij}(t) + M(t) * x_i(t) * y_j(t)) \tag{2.23}$$

  where $M(t)$ is the neuro-modulation function, which can be implemented by a linear neuron, replacing the static update factor γ. In this case, the update rule is slightly different from the previous one, not scaling $Hebb_k(t-1)$ by $(1-\gamma)$ but by clipping the values instead.

- **Eligibility Trace**: This mechanism is inspired by the short-term retroactive effect of neuromodulatory dopamine. In practice, this eligibility trace creates a fast-decaying potential weight change, keeping a memory of which synapse contributed to recent activity. To implement this mechanism, the following formula is proposed:

$$Hebb_{ij}(t+1) = Clip(Hebb_{ij}(t) + M(t) * E_{ij}(t)) \tag{2.24}$$

$$E_{ij}(t+1) = (1-\eta)E_{ij}(t) + \eta x_i(t) y_j(t) \tag{2.25}$$

  where $E_{ij}(t)$ is the exponential average of the Hebbian updates, and η is a trainable parameter, allowing for even more flexibility in the method.

These new proposals pushed even further the applicability of the "Learning to learn" framework, this time achieving better results with respect to back-propagation on another set of tasks such as language modeling, a very hot topic nowadays, achieving lower perplexity.

2.4.4 Limitations

The new framework addresses many of the problems associated with linear neural networks; however, this comes at the cost of a significant increase in the number of parameters. Consider a weight matrix $W \in \mathbb{R}^{N \times M}$ with a total of $N \times M$ parameters. For the base version of the framework, we have the Hebbian parameters $\alpha \in \mathbb{R}^{N \times M}$ and the Hebbian trace $Hebb(t) \in \mathbb{R}^{N \times M}$. This results in doubling the number of trainable parameters and tripling the parameters used during inference. In the neuromodulated version with the eligibility trace, we must also account for the parameters of $M(t) \in \mathbb{R}^N$ and the non-trainable eligibility trace $E(t) \in \mathbb{R}^{N \times M}$, resulting in more than four times the memory requirement of a vanilla linear layer.

Another point to consider is that in much of the research, Hebbian plasticity is somewhat kept separate from the network weights, as seen in Formula 2.22, where the output is the sum of the "normal" part of the network, $w_k x_k$, and the plastic part $a_k Hebb_k$. A "Hebbian-only" approach, where the Hebbian rule directly updates the weights, still has to be explored.

Additionally, a limitation of Miconi's work is the exclusive use of single-layered neural networks. The application of these principles to deep networks remains unexplored. This gap presents a very interesting starting point for the main topic of this thesis.

3 Motivation

In the previous chapter, an extensive literature review was performed, highlighting the main pain points and strengths of the most promising works. This serves as the starting point for the novel proposal of this thesis, which will be detailed in the next chapter.

3.1 Parameter reduction

The primary motivation for this thesis is the significant reduction in the number of parameters proposed by Ferigo et al. [9], which achieves a reduction of approximately 97% while maintaining the same level of performance. Reducing the number of parameters to optimize is crucial because it offers several key advantages:

- **Reduced Memory Requirements**: It is straightforward to understand how such a reduction allows smaller devices to run AI models, particularly in embedded systems where resources are extremely limited and precious. By decreasing the memory footprint, more compact and efficient devices can leverage advanced AI capabilities without the need for extensive hardware upgrades.
- **Faster Convergence Properties**: Having fewer parameters to optimize translates to solving smaller problems. This is critical as it demands fewer resources during training, especially considering that the most commonly used optimizers today are based on the Adam algorithm [19], which increases memory requirements up to four times the number of parameters to optimize. Therefore, fewer parameters mean faster training times and reduced computational overhead.
- **Environmental Implications**: Machine learning models typically require substantial computational power, leading to significant energy consumption and a corresponding environmental impact. Reducing the number of parameters has a cascade effect: it lowers the memory usage and computational power needed, thereby directly decreasing energy consumption. This reduction is beneficial during both the training phase, which can be extremely resource-intensive, and the inference phase, which involves deploying the model for real-world use. Lower energy consumption not only reduces operational costs but also minimizes the carbon footprint associated with running large-scale machine learning models. In an era where climate change is a big global issue, creating more energy-efficient models contributes to the broader effort of reducing greenhouse gas emissions. Thus, parameter reduction aligns the development of AI technologies with sustainable practices, promoting a greener and more environmentally responsible approach to this field.

In this thesis, I will employ this novel concept of "neuron-centric" Hebbian Learning, utilizing only one parameter per neuron. This approach not only exemplifies the benefits outlined above but also shows a way for more efficient and sustainable machine learning models.

3.2 Learning to learn

The second key point is inspired by the interesting research conducted by Miconi [30], which, for the first time, used backpropagation to train Hebbian parameters, enabling plasticity in neural networks across a wide variety of tasks. Despite its innovative approach, this work has been criticized in the previous section for its significantly higher number of parameters to train and for maintaining a separation between the Hebbian and vanilla components of a linear layer. However, theoretically, using backpropagation to train such Hebbian parameters presents a better alternative to the use of evolutionary algorithms, which can require an impractically large amount of memory as the network scales up in size.

My approach will leverage this novel solution to eliminate the need for evolutionary algorithms. This is achieved by integrating a "full-Hebbian" approach, in contrast to Miconi's method, which separates Hebbian and standard components within the linear layers.

3.3 Contrastive Learning

This third domain, which may initially appear unrelated to the topics discussed so far, is actually a crucial factor in developing the new Hebbian rule proposed in this thesis.

To provide some context, I conducted extensive experiments with nearly all the Hebbian rules proposed in various works, including the vanilla Hebbian rule, Oja's variant, the ABCD rule, WTA (Winner-Take-All), and Soft WTA. Unfortunately, none of these methods were able to converge sufficiently to have competitive results, even for smaller tasks, when compared to existing literature. However, other approaches, such as the Forward-Forward method proposed by Hinton [14], have demonstrated the ability to achieve competitive results by employing a contrastive learning setting. This method, as previously discussed, involves the use of positive and negative samples and employs a complex approach to predict labels through multiple forward passes.

The Forward-Forward method's success highlights the potential of contrastive learning in enhancing the performance of neural networks in a biologically plausible scenario. Contrastive learning focuses on learning representations by contrasting positive and negative pairs of examples, which encourages the model to differentiate between similar and dissimilar instances.

3.3.1 The intuition

Hinton [14] says that using a loss function and backpropagating only through a specific layer [14], where the derivative depends solely on pre and post-synaptic activation, does not compromise biological plausibility. This suggests that any loss applied at the layer level is inherently biologically plausible.

The question that arises is then: what should a layer do to generate sparse and meaningful representations that are distinct across different classes? The solution may lie in applying a loss that maximizes the distance between samples from different classes while minimizing it for samples within the same class. By doing so, the layer can effectively discriminate: given certain features, the layer learns which outputs should be similar to those of other samples and which should differ, thus enabling learning at the layer level.

3.4 The algorithm

While the technical details will be elaborated in the next chapter, here is a summary of what the novel algorithm will do and its theoretical advantages. This algorithm integrates all three main frameworks discussed so far: it will use back-propagation to train the Hebbian parameters, these parameters will be neuron-centric, and the Hebbian rule employed will be the one described below. The use of the first two frameworks will allow summing the advantages of both methods while overcoming their criticized pain points: the huge amount of parameters used in the work of Miconi [30] is reduced by the neuron-centric approach and the evolutionary search of the latter is overcome by the use of backpropagation.

In addition to the previously mentioned benefits, this method aims for simplicity. Specifically, the rules can be applied to both linear and convolutional layers, making it versatile regardless of architectural choices. Furthermore, it maintains the ease of use characteristic of standard machine learning models in an end-to-end training framework. This ensures that the method is not only theoretically robust but also practical and straightforward to implement in various neural network architectures. This rule will be applicable not only to linear layers but even to convolutional ones.

4 Method

In this chapter, all the mathematical details of this algorithm's proposal are presented along with the motivation behind each choice.

4.1 The contrastive learning rule

As explained in the previous chapter, the idea is to have a loss function that can minimize the similarity between similar samples and maximize it for different samples. To calculate the similarity between samples, I employed cosine similarity, which is the cosine of the angle between two vectors. It ranges from values $[-1, 1]$, where a value of -1 means that the two vectors are opposite, 0 stands for orthogonality, and 1 for equality. The formula is the following:

$$\cos(\theta) = \frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\|\|\mathbf{B}\|} = \frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} A_i^2} \cdot \sqrt{\sum_{i=1}^{n} B_i^2}} \tag{4.1}$$

where $\mathbf{A}$ and $\mathbf{B}$ are the vectors for which we want to calculate the similarity. To be able to calculate the cosine similarity, the data needs to be presented in batches during the training phase. However, this is not a problem: it generally does not make sense to train a model element by element, except for particular cases, and doing so is significantly slower. This way, the cosine similarity between all the samples in the batch can be easily calculated via matrix multiplication.

Let $\mathbf{Y} \in \mathbb{R}^{B \times N}$ be the output tensor of a layer in a neural network, where B is the batch size and N is the size of the output dimension.

$$\mathbf{Y} = \phi\left(\mathbf{X}\mathbf{W} + \mathbf{b}\right) \tag{4.2}$$

where ϕ is the activation function, specifically the tanh activation function will be employed. The first

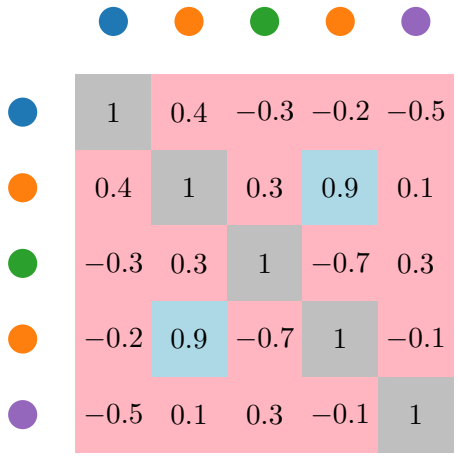


Figure 4.1: The similarity matrix between the batch of samples. On the diagonal the cosine similarity between the same sample, that are not considered and will be set to zero; in red all the pairs that are of different classes (negative samples), and in blue pairs of the same classes(positive samples)

step is the normalization of the output vector to simplify the calculation of the cosine similarity:

$$\mathbf{Y}_{norm} = \frac{\mathbf{Y}}{\|\mathbf{Y}\|_2 + \epsilon} \tag{4.3}$$

where $\|\cdot\|_2$ denotes the Euclidean norm, and $\epsilon = 1 \times 10^{-8}$ is a small constant to avoid division by zero. Calculate the cosine similarity matrix and remove the diagonal:

$$\mathbf{S} = \mathbf{Y}_{norm}\mathbf{Y}_{norm}^T - I. \tag{4.4}$$

At this point, the matrix $\mathbf{S}$ will be in the domain $\mathbb{R}^{B\times B}$ and it will store all the cosine similarity values for each pair of samples in the batch. The diagonal elements are set to zero because we know that the cosine similarity between the same element will always be one and we do not want to consider them in the final loss calculation.

Let us now consider $\mathbf{t} \in \mathbb{R}^B$ as the set of labels of the samples in the batch. To discriminate between positive and negative examples, we can use that vector to create a positive mask $\mathbf{M}_{\text{pos}} \in \mathbb{R}^{B\times B}$ and a negative mask $\mathbf{M}_{\text{neg}} \in \mathbb{R}^{B\times B}$:

$$(\mathbf{M}_{\text{pos}})_{ij} = \begin{cases} 1 & \text{if } \mathbf{t}_i = \mathbf{t}_j \text{ and } i \neq j \\ 0 & \text{otherwise} \end{cases}, \tag{4.5}$$

$$(\mathbf{M}_{\text{neg}})_{ij} = \begin{cases} 1 & \text{if } \mathbf{t}_i \neq \mathbf{t}_j \text{ and } i \neq j \\ 0 & \text{otherwise} \end{cases}. \tag{4.6}$$

Note that both matrices will have a diagonal set to zero, in order to not consider the same element in the loss. This double operation to set the diagonal to zero is redundant; however, for clarity, it is reported twice here, while in the code it will be used in a more compact calculation. In Fig. 4.1 an illustration of how the similarity matrix and the masks interact with each other.

Now that we have all the necessary components, we can calculate the loss for the positive and negative examples:

$$\mathcal{L}_{pos} = \mathbf{M}_{\text{pos}} \cdot (1 - \mathbf{S}). \tag{4.7}$$

Multiplying $\mathbf{M}_{\text{pos}}$ by $(1 - \mathbf{S})$ shifts the domain of S to $[0, 2]$. If two positive samples are very similar, their cosine similarity will be close to 1. Applying that formula shifts the value close to zero, meaning that the loss is small and the layer is working well, thus it does not need contribution from that pair

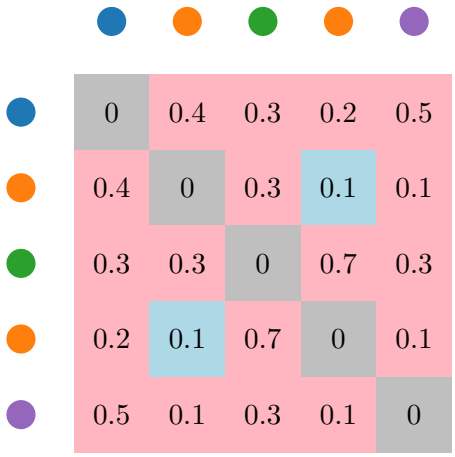


Figure 4.2: The resulting similarity matrix to be minimized, where the cosine similarity is set to zero on the diagonal. The absolute value of the similarity is considered for the negative samples, and the shifted cosine similarity is used for the positive samples.

of samples. Otherwise, if the pair of positive samples is very different, the cosine similarity will have a value of -1 and thus a shifted value of 2, which will give more contribution to the loss to fix this inequality.

$$\mathcal{L}_{neg} = \mathbf{M}_{\text{neg}} \cdot |\mathbf{S}|. \tag{4.8}$$

Instead of shifting the cosine similarity domain, here I decided to use the absolute value of the cosine similarity matrix in order to promote a cosine similarity of zero that will lead to orthogonal representation and not opposite ones. This is beneficial because in the batch, the majority of the pairs will be negative samples, and pushing to opposite directions may obstruct the loss from converging to a good solution. The final loss is then the following:

$$\mathcal{L} = \mathcal{L}_{neg} + \mathcal{L}_{pos}. \tag{4.9}$$

In Fig. 4.2 the loss is represented as a matrix where each element has to be minimized. In the ideal scenario where all the values are zero we have all the negative samples orthogonal between each other and all the positive samples equal to each other.

4.1.1 Derivation of the Loss

Once we have the loss function, the derivative with respect to the weights of the linear layer should be calculated to update the weights using a Hebbian update. This can be achieved using the chain rule up to the weights of the layer. The first step is to calculate the derivative of the loss with respect to the similarity matrix S:

$$\frac{\partial \mathcal{L}}{\partial S} = \frac{\partial \mathcal{L}_{\text{neg}}}{\partial S} + \frac{\partial \mathcal{L}_{\text{pos}}}{\partial S}. \tag{4.10}$$

For the negative part of the loss:

$$\frac{\partial \mathcal{L}_{\text{neg}}}{\partial S} = \frac{\mathbf{M}_{\text{neg}} \cdot |\mathbf{S}|}{\partial S} = \mathbf{M}_{\text{neg}} * \mathbf{S}.\text{sign}(). \tag{4.11}$$

For the positive part of the loss:

$$\frac{\partial \mathcal{L}_{\text{pos}}}{\partial S} = \frac{\mathbf{M}_{\text{pos}} \cdot (1 - \mathbf{S})}{\partial S} = \frac{\mathbf{M}_{\text{pos}} - \mathbf{M}_{\text{pos}} \cdot \mathbf{S}}{\partial S} = -\mathbf{M}_{\text{pos}}. \tag{4.12}$$

Combining these results:

$$\frac{\partial \mathcal{L}}{\partial S} = \frac{\partial \mathcal{L}_{\text{neg}}}{\partial S} + \frac{\partial \mathcal{L}_{\text{pos}}}{\partial S} = \mathbf{M}_{\text{neg}} * \mathbf{S}.\text{sign}() - \mathbf{M}_{\text{pos}} \tag{4.13}$$

where sign() is the sign operator that returns 1 if the value is positive and -1 otherwise. The following step is to calculate the partial derivative with respect to $\mathbf{Y}_{\text{norm}}$:

$$\frac{\partial \mathcal{S}}{\partial \mathbf{Y}_{\text{norm}}} = \frac{\mathbf{Y}_{\text{norm}} \mathbf{Y}_{\text{norm}}^T - \mathbf{I}}{\partial \mathbf{Y}} = \frac{\mathbf{Y}_{\text{norm}} \mathbf{Y}_{\text{norm}}^T}{\partial \mathbf{Y}} - \frac{\mathbf{I}}{\partial \mathbf{Y}} = \mathbf{Y}_{\text{norm}} + \mathbf{Y}_{\text{norm}}^T. \tag{4.14}$$

Next, we consider the derivative of the normalization step and the derivative of $\mathbf{Y}_{\text{norm}}$ with respect to $\mathbf{Y}$ is given by:

$$\frac{\partial \mathbf{Y}_{\text{norm}}}{\partial \mathbf{Y}} = \frac{\partial \left(\frac{\mathbf{Y}}{\|\mathbf{Y}\|} \right)}{\partial \mathbf{Y}}. \tag{4.15}$$

Using the quotient rule we get:

$$\frac{\partial \left(\frac{\mathbf{Y}}{\|\mathbf{Y}\|} \right)}{\partial \mathbf{Y}} = \frac{\|\mathbf{Y}\| \mathbf{I} - \mathbf{Y} \frac{\partial \|\mathbf{Y}\|}{\partial \mathbf{Y}}}{\|\mathbf{Y}\|^2}. \tag{4.16}$$

Since $\|\mathbf{Y}\| = \sqrt{\mathbf{Y}^T \mathbf{Y}}$, its derivative with respect to $\mathbf{Y}$ is:

$$\frac{\partial \|\mathbf{Y}\|}{\partial \mathbf{Y}} = \frac{\mathbf{Y}}{\|\mathbf{Y}\|}. \tag{4.17}$$

Therefore, substituting this back into our expression:

$$\frac{\|\mathbf{Y}\|\mathbf{I} - \mathbf{Y}\mathbf{Y}^T_{\text{norm}}}{\|\mathbf{Y}\|^2} = \frac{1}{\|\mathbf{Y}\|}\left(\mathbf{I} - \mathbf{Y}_{\text{norm}}\mathbf{Y}^T_{\text{norm}}\right). \tag{4.18}$$

Thus, the final derivative for the normalization part is:

$$\frac{\partial \mathbf{Y}_{\text{norm}}}{\partial \mathbf{Y}} = \frac{1}{\|\mathbf{Y}\|}\left(\mathbf{I} - \mathbf{Y}_{\text{norm}}\mathbf{Y}^T_{\text{norm}}\right). \tag{4.19}$$

The next step in the chain of functions is to calculate the derivative of the activation function, specifically the tanh function:

$$\frac{\partial \mathbf{Y}}{\partial \mathbf{Y}_{\text{raw}}} = \frac{\partial \tanh(\mathbf{Y}_{\text{raw}})}{\partial \mathbf{Y}_{\text{raw}}} = 1 - \tanh^2(\mathbf{Y}_{\text{raw}}) \tag{4.20}$$

where $\mathbf{Y}_{\text{raw}}$ is the output of the linear layer before applying the activation function. The final step consists of calculating the partial derivative with respect to the weights:

$$\frac{\partial \mathbf{Y}}{\partial \mathbf{W}} = \frac{\partial\left(\mathbf{X}\mathbf{W} + \mathbf{b}\right)}{\partial \mathbf{W}} = \mathbf{X}. \tag{4.21}$$

Now that we have all the partial derivatives, we can concatenate them via the chain rule:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}} = \frac{\partial \mathcal{L}}{\partial S} \cdot \frac{\partial \mathcal{S}}{\partial \mathbf{Y}_{\text{norm}}} \cdot \frac{\partial \mathbf{Y}_{\text{norm}}}{\partial \mathbf{Y}} \cdot \frac{\partial \mathbf{Y}}{\partial \mathbf{Y}_{\text{raw}}} \cdot \frac{\partial \mathbf{Y}}{\partial \mathbf{W}}. \tag{4.22}$$

In this work the bias term is not considered, however, its derivation is trivial:

$$\frac{\partial \mathbf{Y}}{\partial \mathbf{b}} = \frac{\partial\left(\mathbf{X}\mathbf{W} + \mathbf{b}\right)}{\partial \mathbf{b}} = 1. \tag{4.23}$$

And consequently, the derivative of the bias with respect to the loss is:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{b}} = \frac{\partial \mathcal{L}}{\partial S} \cdot \frac{\partial \mathcal{S}}{\partial \mathbf{Y}_{\text{norm}}} \cdot \frac{\partial \mathbf{Y}_{\text{norm}}}{\partial \mathbf{Y}} \cdot \frac{\partial \mathbf{Y}}{\partial \mathbf{Y}_{\text{raw}}} \cdot \frac{\partial \mathbf{Y}}{\partial \mathbf{b}}. \tag{4.24}$$

4.1.2 The contrastive rule applied to convolution

At first glance, applying the proposed rule to a convolutional layer may seem challenging due to the nature of convolutional operations, where a shared convolutional kernel passes through the image to create new output channels, with each kernel generating a new channel appended to the output. However, this approach is still feasible by flattening the output of a convolutional kernel.

Let $\mathbf{Y} \in \mathbb{R}^{B\times C\times W\times H}$ be the output tensor of a convolutional layer in a neural network, where B is the batch size, C is the number of output channels, and W and H are the width and height, respectively. We can flatten the dimensions C, W, and H to obtain $\mathbf{Y} \in \mathbb{R}^{B\times(C\cdot W\cdot H)}$, enabling us to apply the same procedure as described previously.

When calculating the derivative, we follow the same steps, leveraging the versatility of the chain rule. The only difference arises in the calculation of the partial derivative of the weights $\frac{\partial \mathbf{Y}}{\partial \mathbf{W}}$.

Since we are dealing with a convolution, we calculate the partial derivative of the loss with respect to the kernel weights ($\frac{\partial \mathcal{L}}{\partial \mathbf{W}}$) by convolving the original input ($\mathbf{X}$) with the partial derivative of the loss with respect to the output ($\frac{\partial \mathcal{L}}{\partial \mathbf{Y}}$). A visual example is shown in Fig. 4.3. Intuitively, this concept is straightforward, but it becomes less trivial when other parameters are involved in the convolution, such as stride, dilation, grouping, and padding. To be precise, the dilation should be used as stride and vice versa during the backward operation, and additional considerations must be made to ensure the correct handling of these parameters. Fortunately, the PyTorch library provides a function that automatically deals with all these hyperparameters: `torch.nn.grad.conv2d_weight()`.

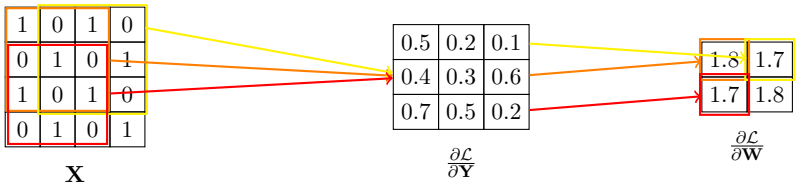


Figure 4.3: Example of the chain rule applied in a Convolutional Neural Network: the input is convolved with the partial derivative of the loss with respect to the output of the convolution operation.

4.1.3 Label smoothing

The proposed loss aims to maximize the separation between different samples. In an ideal scenario where all the input data is perfectly separable, this approach works very well. However, in practice, samples from different classes may share common features, making perfect separation challenging. Szegedy et al. [43] introduced a new regularization technique called "Label Smoothing" to address this issue. Label smoothing accounts for uncertainty in the ground truth by modifying the one-hot encoding in scenarios where the cross-entropy loss is employed, thereby adding uncertainty to the labels. In this way, we can tell to our rule that some negative samples may share a certain degree of features. Label smoothing modifies the target labels $\mathbf{t}$ by distributing a small portion of the label's probability mass to all classes, thereby preventing the model from becoming overly confident. The formula for label smoothing is:

$$\mathbf{t}'_i = (1-\alpha)\mathbf{t}_i + \frac{\alpha}{K} \tag{4.25}$$

where $\mathbf{t}_i$ is the original one-hot encoded label for class i, $\mathbf{t}'_i$ is the smoothed label for class i, α is the smoothing parameter (a small value, e.g., 0.1), K is the number of classes. Incorporating label smoothing into the proposed rule may help the model generalize better by allowing common features to arise in the output data, thus making it more robust and achieving higher performance. Label smoothing is introduced into the algorithm by modifying the similarity masks. Specifically, the positive and negative similarity masks are adjusted as follows:

$$\mathbf{M}'_{pos} = (1-\alpha)\mathbf{M}_{pos} + \frac{\alpha}{B-1} - I\frac{\alpha}{B-1}, \tag{4.26}$$

$$\mathbf{M}'_{neg} = (1-\alpha)\mathbf{M}_{neg} + \frac{\alpha}{B-1} - I\frac{\alpha}{B-1} \tag{4.27}$$

where B is the batch size, representing the number of elements in the batch; I is the identity matrix and $I\frac{\alpha}{B-1}$ is needed to zero out the element in the diagonal. Note that the use of $B-1$ is due to the fact that for each element in the batch, the similarity with itself is not considered in label smoothing and thus reducing the total number of elements to $B-1$ This adjustment ensures that each element in the similarity masks is smoothed by distributing a small portion of the probability to all elements, reducing overconfidence and enhancing generalization. Note that with such modification the formula of the derivative remains unchanged.

4.1.4 Rule on the final layer

The rule proposed so far works well for the hidden layers, allowing the network to learn new representations of the data as it gets deeper. However, in the last layer, there should be a mapping between the output and the predicted label. Therefore, it should learn an exact order for the output to make predictions, and thus a different rule should be used in this case. Note that using normal backpropagation only for the last layer does not break biological plausibility, as already mentioned in the related work chapter. However, to maintain a neuro-Hebbian update even in the last layer of the network, the Hebbian descent rule explained by the formula 2.8 is employed. Since the loss used at the top of the network is the cross-entropy, the mean value is considered zero, and thus no normalization is employed. The final formula is as follows and it is equal to the derivative of the cross-entropy loss:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}} = \mathbf{X}(\mathbf{T} - \mathbf{Y}) = \mathbf{X}(\mathbf{T} - \text{softmax}(\mathbf{XW} + \mathbf{b})). \tag{4.28}$$

As mentioned earlier, the bias term is not considered in the experimental part, but for completeness, the bias term formula is provided:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{b}} = \mathbf{T} - \mathbf{Y} = \mathbf{T} - \text{softmax}(\mathbf{XW} + \mathbf{b}). \tag{4.29}$$

4.2 Neuron-centric approach

In the previous section, we proposed a new rule, and we now have all the formulas needed to calculate the $\Delta\mathbf{W}$ for all the layers in the network, whether they are linear or convolutional layers. The original approach proposed by Ferigo et al. [9] employed the ABCD rule. However, in this case, we will consider our rule in the form of the vanilla Hebbian rule, requiring only the hyperparameters corresponding to $\mathbf{C}$ in the ABCD rule. This is motivated by the fact that my rule depends on both pre and post-synaptic values that are needed in order to calculate the local derivative only and thus the other hyperparameters are not needed, and no bias will employed. Specifically, the update rule will be as follows:

$$\Delta w_{ij} = \eta C_i C_j \frac{\partial \mathcal{L}}{\partial w_{ij}} \tag{4.30}$$

where η is the initial learning rate, set globally for the network, C_i is the pre-synaptic value of the i^{th} neuron, and C_j is the post-synaptic value of the j^{th} neuron. Note that the partial derivative of the loss refers to the local rule and not to the cross-entropy applied on top of the network.

The original version of the algorithm expects that each neuron's parameter, only C in this case, is shared across all synapses it is involved in. For instance, the value C of neuron k will be used in the update rule for the weights where that neuron is a post-synaptic neuron and in the update rule where it is pre-synaptic. A linear layer is nothing more than a matrix representing all the synaptic weights between a set of pre and post-synaptic neurons, but these neurons, except for the last set of neurons, should share the C parameters with the previous and subsequent layer matrices. We propose a slightly different approach where each neuron has two parameters: one for when it is an input and one for when it is an output, to better fit with traditional machine learning workflows. Specifically, I assign two different C values to each neuron: one for presynaptic activations and one for postsynaptic activations. This modification allows the parameters for each linear layer to be set independently, avoiding the need to share data between layers and thereby reducing complexity. For example, a PyTorch linear layer can be extended by adding C_i and C_j parameters for input and output dimensions respectively, without needing to share the C_j parameter with the subsequent layer. Although this approach nearly doubles the number of parameters to train compared to the original neuron-centric algorithm, it remains efficient because the neuron-centric approach uses significantly fewer parameters compared to a synaptic-centric approach as discussed in Section 2.1.7. Figure 4.4 shows how the two values, C_i and C_j, are assigned to a neuron.

Figure 4.4: A representation of weight sharing in the left figure, where the same C value is used for both pre-synaptic and post-synaptic activations. In contrast, the right figure shows how this sharing property is broken by using separate values C_{pre} and C_{post} for pre-synaptic and post-synaptic activations, respectively.

At the end of the process, we will have a matrix C that is the outer product of all $C_{pre} \in \mathbb{R}^I$ and $C_{post} \in \mathbb{R}^O$ values, resulting in a matrix with the same dimensionality as the weight matrix:

$$C = C_{pre} \otimes C_{post} \in \mathbb{R}^{I \times O}. \tag{4.31}$$

The formula to update the weights in matrix notation will be as follows:

$$\Delta \mathbf{W} = \eta C \frac{\partial \mathcal{L}}{\partial \mathbf{W}}. \tag{4.32}$$

In convolutional layers, the weights are organized differently: they are composed of multidimensional kernels where $K \in \mathbb{R}^{I \times O \times H_k \times W_k}$, with I and O representing the number of input and output channels, respectively, and H_k and W_k representing the height and width dimensions of the kernel, respectively. To accommodate this multidimensionality, we have two choices:

- **Flattening post-synaptic**: By flattening the kernel weights, we can represent K as $K \in \mathbb{R}^{I \times O \cdot H_k \cdot W_k}$, which has two dimensions similar to the linear layer setting. This approach allows for more similarity with the linear layer but requires a reshaping operation to produce a single vector for each dimension for calculating the outer product. Moreover, this solution requires storing more parameters, specifically a total of $O \cdot H_k \cdot W_k$ parameters.

- **Keeping the dimensionality**: We can use three different vectors to store the C values for each dimension: one for the output dimension, one for the kernel width, and one for the kernel height. By concatenating these outer products, we can maintain the same dimensionality as the output filter. In this case, the number of parameters is reduced, with a total of $O + H_k + W_k$ parameters.

Using these methods ensures that our approach is adaptable to the structure of convolutional layers while retaining the benefits of the neuron-centric update rules. In particular, the second method will be employed because it uses fewer parameters and aligns better with the neuron-centric idea. Let $C_{\text{post}} \in \mathbb{R}^O$, $C_{\text{height}} \in \mathbb{R}^{H_k}$ and $C_{\text{width}} \in \mathbb{R}^{W_k}$ the vectors representing the post-synaptic values in a convolutional layer, the final formula to calculate C is as follows:

$$C = C_{\text{pre}} \otimes C_{\text{post}} \otimes C_{\text{height}} \otimes C_{\text{width}} \in \mathbb{R}^{I \times O \times H_k \times W_k}. \tag{4.33}$$

4.3 Backpropagation

In the previous two sections, I provided a comprehensive description of the proposed rule, and here I will describe how the Hebbian hyperparameters will be trained. Inspired by the works of Miconi [30], I will employ backpropagation to train the Hebbian parameters. The parameter η is a hyperparameter equivalent to an initial learning rate. As we will see, the C parameters will be initialized to ones, making an initial learning rate necessary to avoid numerical instability.

At first glance, training these parameters with backpropagation may appear straightforward; however, this is not the case here. Let's start with the simplest case and consider a shallow network with one layer applying the proposed rule for the last layer. We have our input data $\mathbf{X}$, which is fed to the network. The model processes this data and returns an output $\mathbf{Y}$. This output is then used to calculate the change in weights $\Delta \mathbf{W}$. However, the output $\mathbf{Y}$ is not immediately influenced by $\Delta \mathbf{W}$. Note that the gradient should be calculated based on the change in weights $\Delta \mathbf{W}$. This poses a challenge to the use of backpropagation to train such parameters. Two solutions can be applied to overcome this issue:

- **Double forward pass**: the first pass is to calculate $\Delta \mathbf{W}$, and the second pass returns $\mathbf{Y}$ after the weight change is applied. This allows the gradient with respect to the C parameters to be calculated via the chain rule. However, this solution requires more computational resources because it involves a double forward pass, which could be avoided as explained in the following point. In the upper part of the Fig. 4.5 the steps for this solution are shown.

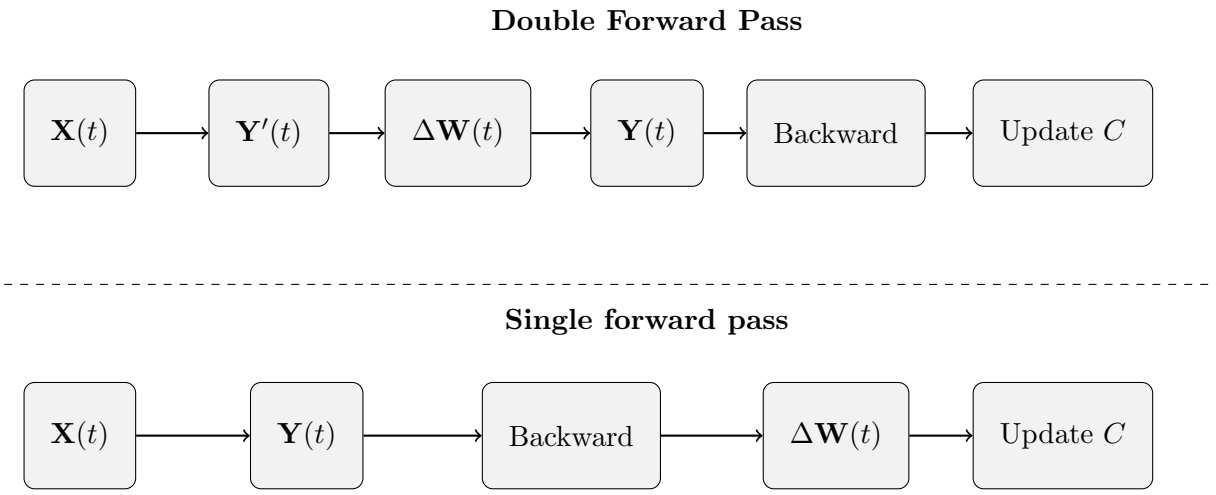


Figure 4.5: On the upper part of the image the process of using a double forward pass is presented: $\mathbf{Y}'$ is the output of the network used for calculate $\Delta\mathbf{W}$ and $\mathbf{Y}$ is the output after applying the change in weights. On the lower part, the steps to allow a single forward pass are presented.

- **Single forward pass**: The second solution is to skip the gradient update for the first iteration. In this way, at the second iteration, the weights are already modified by $\Delta\mathbf{W}$, and the gradient on the loss of the current batch with respect to the previous weight update can be calculated. Unfortunately, the `autograd` library of PyTorch is unable to correctly work with this setting: when we use the `loss.backward()` function on the loss, `optimizer.zero_grad()` should be called to zero out the `.grad` property of the tensors involved in the loss calculation. However, it also erases the `.grad_fn` property that contains the graph of the operations performed, thus making it impossible to apply the chain rule and get the gradient update. A solution to this problem is to separate the normal forward pass, that calculates both $\mathbf{Y}$ and $\Delta\mathbf{W}$, from the weight change step and perform the latter operation only after `optimizer.zero_grad()` is called. This way, at every subsequent step, the weight matrix $\mathbf{W}$ contains in its graph the contribution of $\Delta\mathbf{W}(t-1)$ from the previous batch of samples. The autograd functionality in PyTorch can then correctly find the gradient with respect to all the C parameters. In the lower part of Fig. 4.5, the steps for this solution are shown. This solution involves temporarily saving the values of the input and output of each layer to be available for the step that calculates the change in weights, requiring additional memory in the training phase.

Both solutions have their drawbacks. The first solution requires a double forward pass, while the second solution necessitates temporarily storing the input and output values for every layer. Specifically, in the latter case, the memory requirement, expressed as the number of additional parameters, given by:

$$\text{Memory Requirement} = B \times \sum_{l=1}^{L} (I_l + O_l) \tag{4.34}$$

where I_l and O_l are the input and output dimension of that layer l, respectively and L is the total number of layers in the network; B is the batch size. For a convolutional layer, the formula is the same, with O_l equal to $C_{o_l} \times H_{o_l} \times W_{o_l}$ which are the dimensions of the output of the convolutional layer. However, if the batch size is reasonably small the amount of memory needed is way less than the memory needed to store the whole model.

Despite its increased memory usage, the second method was chosen for this thesis for the following reasons. Treating C as a set of learning rates, a double forward pass would calculate the gradient on $C(t)$ using $\mathbf{Y}(t)$, which is based on $\Delta\mathbf{W}(t)$. This would push C in a direction that speeds up learning

for the current batch $\mathbf{X}$, resulting in higher C values and leading to overfitting. Conversely, with a single forward pass, $\Delta C(t)$ is calculated based on $\Delta \mathbf{W}(t-1)$, ensuring that the change in the learning rate is effective for both the current batch $\mathbf{X}(t)$ and the previous batch $\mathbf{X}(t-1)$.

It is interesting to note that in all the works of Miconi [31], this issue does not exist. The formula used, as shown in 2.22, maintains a Hebbian trace $Hebb_k(t)$ that is updated at each new step t based on the pre and post-synaptic activations. The output $y(t)$ is then calculated by the product of the weights and the inputs, plus the product of the Hebbian trace, the Hebbian parameters, and the inputs. In this case, the output $y(t)$ is immediately influenced by the Hebbian trace, even though it is initially set to zero. Consequently, the `autograd` library can effectively compute the gradient for the learning rates. However, Miconi's approach significantly increases the number of parameters, as discussed in Section 2.4.4, and it also uses backpropagation to train the weights. This deviates from the objective of having only a Hebbian rule update the weights. These are the reasons why the use of a Hebbian trace is not considered in this thesis.

4.3.1 Momentum

As is well-known in the field of Machine Learning, using a pure Stochastic Gradient Descent algorithm is not a common practice. Pure SGD, while simple, can be inefficient due to its high variance, which causes the optimization process to be slow and noisy. This is particularly problematic for deep learning models, where the loss landscape can be highly non-convex with many local minima and saddle points.

To address these issues, a momentum strategy can be employed. Momentum helps accelerate gradient vectors in the right directions, thus leading to faster converging. By dampening oscillations and providing a more stable convergence path, momentum can significantly improve the performance of the network. However, it is important to note that the method works even without momentum, making it suitable for scenarios where minimizing memory usage is crucial. This is because using momentum effectively doubles the amount of memory required, as it involves storing additional momentum values proportional to the number of weights in the network.

Nevertheless, the benefits of using momentum are often worth the additional memory cost, as it can lead to faster convergence and better overall performance of the model. Therefore, in this thesis the version with the momentum strategy is employed, however, this is not a strict requirement and not using it will decrease the performance of only a couple of percent points on accuracy on various tasks.

4.3.2 Memory usage

I have detailed how this new method can be implemented, and in this section, I will perform a theoretical analysis of the overall size of the model during both the training and inference phases. For simplicity, let's consider a network composed of a single layer $W \in \mathbb{R}^{N \times N}$ with N being sufficiently large. It will have pre-synaptic and post-synaptic Hebbian coefficients: $C_i \in \mathbb{R}^N$ and $C_j \in \mathbb{R}^N$. The memory size during inference-only will be $W + 2N$. If we only want to keep the synaptic weights and use the frozen model, we can reduce the number of parameters to W as in a normal neural model. In the training phase, if we consider using an Adam-like [19] optimizer, the memory usage will be four times the number of trainable parameters. In this case, the trainable parameters are only C_i and C_j, so we will have $W + 8N$ memory requirements. If we use the momentum strategy, a tensor with the same size as the weights should be kept in memory, thus requiring $2W + 8N$. If we then consider the double forward pass, we need to add another $2BN$ where B s the batch size. The overall memory requirements will be:

$$2W + 8N + 2BN = 2W + 2N(4 + B) = \mathcal{O}(2W). \tag{4.35}$$

With classical backpropagation, the amount of memory used highly depends on the optimizer. An SGD with momentum only requires $2W$, the same order of magnitude as this new method. If an Adam-like optimizer is employed, the memory consumption grows to $4W$ which is double that of the previous methods.

While the formula 4.35 is theoretically correct, it does not consider the memory usage coming from the chain rule. In fact, because the derivative of Y in respect of C_i and C_j is calculated from the

formula $Y = (W + C_i C_j \Delta W)X$ we have then:

$$\frac{\partial Y}{\partial C_i} = \frac{\partial}{\partial C_i}\left[(W + C_i C_j \Delta W)X\right] = (C_j \Delta W)X, \tag{4.36}$$

$$\frac{\partial Y}{\partial C_j} = \frac{\partial}{\partial C_j}\left[(W + C_i C_j \Delta W)X\right] = (C_i \Delta W)X. \tag{4.37}$$

This means that ΔW should be kept in memory to calculate the gradient on C_i and C_j, adding another N^2 factor. However, things are more complex in practice: the `autograd` module of PyTorch automatically saves in the context all the variables needed, so ΔW will be stored in the context. If the variable already has a reference, it is not cloned, and thus no increase in memory occurs, as is the case with the parameters C_i and C_j. Keeping this in mind, if the output is calculated using momentum: $Y = (W + C_i C_j M_{omentum})X$, that variable is saved in the current layer, and thus no additional memory will be used. Therefore, the formula, for both the variables using momentum and without, has the same memory requirements shown in Formula 4.35. To conclude, this method has competitive memory usage only if compared with memory demanding optimizers but still uses more memory than a SGD with Momentum optimizer.

In a scenario where backpropagation is used to train the Hebbian parameters only in an initial part of the training, the network will then have faster training time by removing the backward pass completely and maintaining only $W + 2N$ memory usage.

4.3.3 Activation function

The choice of activation function is usually driven by various considerations. For example, the ReLU activation function is widely used because it addresses the vanishing gradient problem in deep networks, leading to faster convergence. On the other hand, the tanh activation function is more biologically plausible. However, it can lead to the vanishing gradient problem because its derivative approaches zero for very high or very low values.

In this case, due to the local Hebbian rules, we don't encounter the vanishing gradient problem. Therefore, the tanh function is a better choice for this algorithm because of its biological plausibility. Additionally, tanh allows for better separation properties since it can return negative values, making orthogonality between vectors more easily achievable. In this work, we will use then the tanh activation function.

4.3.4 A note on batch normalization

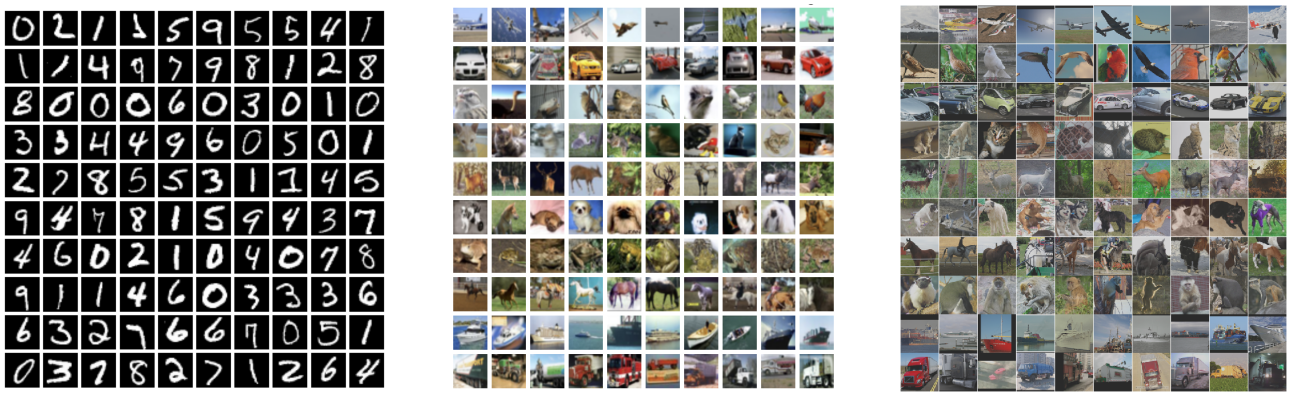
Batch normalization was introduced by Ioffe and Szegedy [16] and the main idea behind it is to tackle the problem of the internal covariate shift. By normalizing the inputs of each layer to have a mean of zero and a variance of one, batch normalization helps to make the learning process more stable. This allows for using higher learning rates and speeds up the training process. Batch normalization does this by calculating the mean and variance of the inputs for each mini-batch and then using these values to normalize the inputs.

However, the term "internal covariate shift" has caused some confusion and debate. Some researchers believe that the improvements seen with batch normalization are not necessarily due to reducing internal covariate shift but because of other factors like added regularization and making the optimization process smoother, adjusting the mean and the standard deviation to a normal distribution for the next input layer.

In my work, I decided not to use batch normalization. The reason is that the rules that modify the layers aim to separate the representations of different samples as much as possible, while the aim of batch normalization is to normalize the input data for a given layer. If we normalize these representations, reducing the standard deviation to 1, we reduce the differences between the classes. Think about the classes as a collection of Gaussian-like distributions spread over a given domain; reducing the standard deviation will bring these distributions closer together, making it more difficult for the next layer to understand the boundaries between them.

5 Experimental setup

In this chapter, I will present the tasks where we tested our proposed method to assess its effectiveness. In particular, I will test it with basic classification tasks: *MNIST*, *CIFAR10*, and *STL10*. Another interesting task that will be considered is Continual Learning, specifically with the SplitMNIST task.



(a) MNIST [25] (b) CIFAR-10 [20] (c) STL-10 [6]

Figure 5.1: Examples of images from the datasets used in the experiments: MNIST (a), CIFAR-10 (b), and STL-10 (c).

5.1 Classification tasks

Classification tasks are a fundamental component of machine learning, where the objective is to assign a label to an input based on its features. These tasks serve as standard benchmarks to evaluate the performance of various algorithms and models. In particular, Image classification is the benchmark to assess the performance on visual tasks. Below, I provide an overview of the datasets used in the classification tasks for this study.

- **MNIST**: The MNIST dataset [25] is a large collection of handwritten digits that is commonly used for training various image processing systems. It consists of 70,000 grayscale images, each with a size of 28x28 pixels. The dataset is split into 60,000 training images and 10,000 test images, with each image labeled from 0 to 9. The simplicity and standardized nature of MNIST make it a popular choice for the initial testing of classification algorithms. Its small image size and limited complexity allow for rapid experimentation and iterative improvements.

- **CIFAR10**: The CIFAR10 dataset [20] is a more challenging classification task compared to MNIST. It contains 60,000 32x32 color images divided into 10 classes. The dataset is split into 50,000 training images and 10,000 test images. The classes include airplanes, automobiles, birds, cats, deer, dogs, frogs, horses, ships, and trucks. The diversity and complexity of the images in CIFAR10 make it a valuable benchmark for evaluating the performance of deep learning models. The relatively small image size and well-defined classes facilitate efficient training and testing while providing a substantial challenge due to the natural variability within each class.

- **STL10**: The STL10 dataset [6] is derived from the ImageNet dataset and is designed for evaluating unsupervised feature learning algorithms. It consists of 96x96 color images and includes 10 classes similar to those in CIFAR10. The dataset contains 5,000 labeled training images, 8,000 test images, and 100,000 unlabeled images. The higher resolution of STL10 images provides a more challenging benchmark for developing scalable learning methods. The large set

of unlabeled images is particularly useful for training models that can leverage unsupervised or semi-supervised learning techniques, which is crucial for improving generalization and performance in real-world applications where labeled data is often limited.

5.2 Continual Learning tasks

Continual Learning is an area of machine learning focused on developing models that can learn continuously from a stream of data without forgetting previous knowledge. This ability is essential for creating adaptive systems that can evolve over time and handle new tasks as they arise. The major problem of networks trained with backpropagation in this field is known as Catastrophic Forgetting [28]: neural networks trained sequentially on different tasks will lose the ability to remember previous tasks and thus fail drastically to solve them.

In the comprehensive survey conducted by Irie et al. [17], it is highlighted that there is a lot of confusion in the field, and many works use datasets that are not ideal for testing continual learning scenarios. They describe the simplest task for Continual Learning, which is the SplitMNIST, where the same output head is used during all tasks. Some works use a different output head for every subtask; however, this approach is considered less challenging and results with a single head are significantly more difficult. In the SplitMNIST task, the MNIST dataset is divided into five binary classification tasks: 0,1, 2,3, 4,5, 6,7, and 8,9. The model is trained sequentially on these tasks, and its performance is evaluated on all tasks at the end of training. This setup highlights the model's ability to retain knowledge and avoid catastrophic forgetting, a common challenge in continual learning scenarios.

5.3 Baselines

In this section, I will outline the baselines used for comparison, which include various state-of-the-art results. These baselines are crucial for evaluating the performance of the proposed method against established techniques in the field.

5.3.1 Linear network

As discussed in the related work section, most studies use shallow networks, with the exception of the work by Moraitis et al. [34], which uses one and two-layer networks of 2000 neurons each. However, their training procedure does not align with standard practices, see the related work section 2.1.6. Their performance reaches $(96.31 \pm 0.06)\%$ and $(97.80 \pm 0.02)\%$ for the one and two-layer networks, respectively, after 100 epochs of training. However, only the two-layer network will be considered as a reference.

Another significant work is the Forward-Forward algorithm [14], which achieves accuracies of 98.6% on MNIST and 56% on CIFAR10. This method employs four hidden layers with 2000 neurons each and uses ReLU activation for the MNIST dataset.

Network setup

For comparison, the proposed linear network will also use four linear layers of 2000 neurons each, with the tanh activation function. It will be trained for 100 epochs, similar to the previous methods. No additional strategies, such as dropout or image augmentations, will be employed in this setup to ensure a fair comparison. The same architecture and training strategy will be used for all the datasets. A comparison with the same network architecture is done with normal backpropagation, in particular, it will employ a momentum optimizer, in order to align with the proposed algorithm. Moreover, different strategies such as the use of label smoothing for the local rule and training without using momentum will be analyzed here. Label smoothing can help prevent the model from becoming overconfident in its predictions, thus potentially improving generalization and robustness. On the other hand, avoiding momentum in training will be considered to evaluate the performance of the model under minimal memory usage conditions.

5.3.2 Convolutional network

In the literature, only the work "Hebbian Deep Learning Without Feedback" [18] effectively uses convolutional networks with multiple convolutional layers, achieving competitive results. Specifically,

they achieved accuracies of 99.4%, 80.3%, 76.2%, and 27.3% on MNIST, CIFAR10, STL10, and ImageNet, respectively.

The comparison against this baseline will be performed using the same architecture used in that work. In particular, the focus will be on the comparison of the two approaches on the CIFAR-10 dataset. I will demonstrate the fragility of that method by changing the activation function and the width factor of the output channel of the network. The width factor, crucial for achieving good performance in their work, was set by default to 4, meaning that at each convolutional layer, the number of output channels is multiplied by that width factor. Reducing this factor significantly decreases the size of the network, especially for deep networks: the number of output channels grows quadratically at each layer.

Network setup

The comparison against this baseline will be done with the same linear and convolutional architectures, for MNIST, CIFAR10, and STL10. Then I will show how fragile is this method, changing the pooling type in the convolutional network and the wide factor of the network that for that work is crucial to determining a good performance; in particular, a factor of 4 is used. When lowered, the performance drastically decreases. Moreover, all experiments with convolutional layers are conducted using multiple linear layers after the convolutions to assess the effectiveness of different types of layers within the same network.

5.3.3 Continual learning

To evaluate the continual learning capabilities, the Single-Head SplitMNIST task will be utilized. This task involves sequentially training a model on different subsets of the MNIST dataset, where each subset contains a pair of digit classes. Specifically, the SplitMNIST task is divided into five distinct tasks, with each task focusing on classifying a pair of digits: (0,1), (2,3), (4,5), (6,7), and (8,9).

The challenge of the Single-Head SplitMNIST task lies in using a single output head for classification across all tasks. This means the model must distinguish between all ten digits despite only being trained on two digits at a time, sequentially. This approach tests the model's ability to retain knowledge from previous tasks (digits) while learning to classify new ones, a key aspect of continual learning. Performance will be evaluated on both linear and convolutional networks.

The continual learning performance will be benchmarked against a baseline model trained with standard backpropagation on the same architecture. By comparing the performance of the continual learning approach to the baseline, we can assess how well the model retains previously learned knowledge and adapts to new tasks without significant forgetting. This comparison will provide insights into the effectiveness of the proposed method in mitigating the catastrophic forgetting problem commonly encountered in such scenarios.

6 Experimental results

In this chapter, all the results of the experiments will be accurately detailed. For every proposed variant, such as the use of label smoothing, the avoidance of momentum, and so on, five runs are executed. The results will consider the average and standard deviation of these runs. For all the runs, both using linear and convolutional networks, a batch size of 32 is employed. Moreover, the Hebbian parameters are all initialized to a value of one. In this context, using weight decay applied to the Hebbian terms, which essentially function as learning rates, is more similar to applying a learning rate schedule than to applying a decay to the network weights. In all the experiments, during the validation and test steps, the weights of the networks remain frozen. The set of weights that performed best on the validation dataset during training is kept and used for evaluation on the test set. While backpropagation is employed in the new proposed method, references to backpropagation in this section correspond to the original backpropagation, without Hebbian rules. The new proposed method will be referred to as Contrastive Hebbian.

6.1 Experiments on a linear network

As discussed before in this case a linear network with 2000 neurons with a tanh activation function is employed. When training with backpropagation the SGD optimizer with momentum is employed, with a learning rate of 0.001. The network trained with my proposal will use the AdamW [26] optimizer with a learning rate of 0.03 and a weight decay of 0.003. Note that the AdamW optimizer modifies only the Hebbian parameters and not the weight of the network. The rationale for employing two different optimizers may seem counterintuitive, however, this approach aligns with the weight update mechanism of the Contrastive Hebbian framework, which uses momentum. The initial learning rate to be applied to the rule is 0.001 and an L2 normalization is applied to the weights at every update rule. This setting is used for all the datasets.

6.1.1 MNIST

In Table 6.1 the results on the test set are presented. As we can see, the Forward-Forward method achieves the highest results. However, the corresponding paper does not specify if this result comes from a single run or a batch of runs. Moreover, the backpropagation baseline was not optimized to achieve the best performance possible in this case and it uses an SGD with momentum optimizer: it could achieve a higher accuracy with an Adam optimizer with a better hyperparameter tuning.

The new method proposed achieves an accuracy of almost 97% (96.98%) and is only 1.12 percentage

Table 6.1: Test set accuracy for the Contrastive Hebbian (CH) method on the MNIST dataset, its variants with label smoothing and without momentum, vanilla backpropagation, the Forward-Forward algorithm, and the SoftHebb variants.

Method	**Accuracy (%)**	Standard deviation
CH	96.98%	± 0.11
CH + Label Smoothing	96.87%	± 0.21
CH without Momentum	95.83%	± 0.15
Backpropagation	98.10%	± 0.06
SoftHebb, 1 layer [34]	96.31%	± 0.06
SoftHebb, 2 layers [34]	97.80%	± 0.02
Forward-Forward [14]	**98.63**%	-

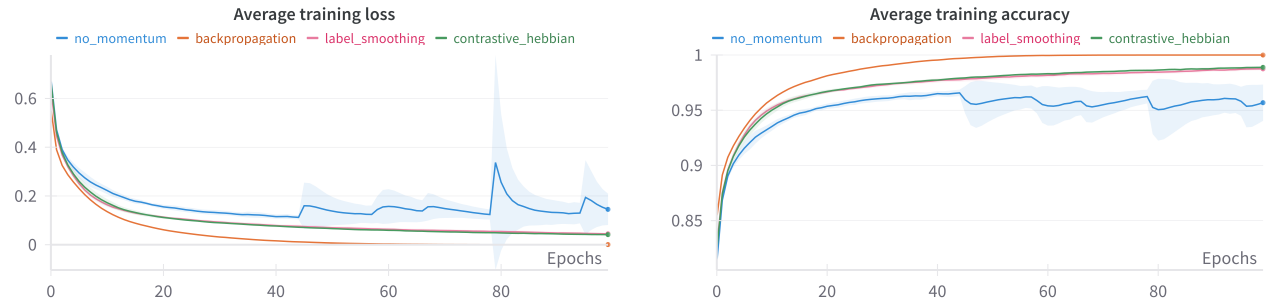


Figure 6.1: Average training loss and accuracy during the training phase over 5 independent runs on the MNIST dataset for linear networks.

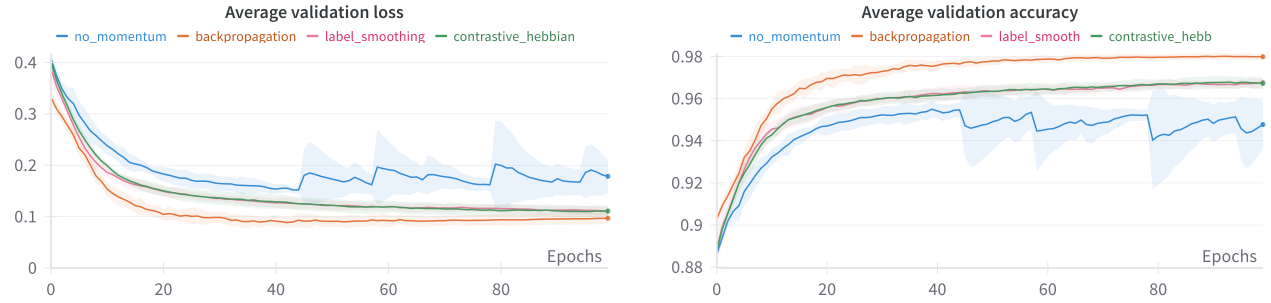


Figure 6.2: Average validation loss and accuracy during the training phase over 5 independent runs on the MNIST dataset for linear networks.

points away from the backpropagation baseline, suggesting that this method is able to converge to a good solution. The Soft-Hebb network achieves higher results only for the 2-layered version, as well as the Forward-Forward method. Despite the slightly lower performance, the main point of this new method is not to outperform other methods but to be competitive while providing an easier, more robust, and extensible method for applying Hebbian learning to deep neural networks. It aims to be more similar to standard approaches without complicated and uncommon strategies.

The variant using label smoothing did not perform better than the Contrastive Hebbian base method, suggesting that this approach does not help convergence as initially hypothesized. However, the performance of the two methods is very similar.

The variant that ran without storing a momentum vector surprisingly did not perform as poorly as expected, with a difference of only 1.15 percentage points from the original version. This suggests that in scenarios where memory requirements are very strict, this variant can be an effective solution. In Fig. 6.1 and Fig. 6.2, we can see its training curves (in blue), and it is clear that at a certain point, this variant started to show some instability. This could have been avoided or minimized by using smaller learning rates or higher weight decay. However, for comparison, the same hyperparameters are used in all the runs for coherence.

6.1.2 CIFAR-10

The tests on the CIFAR-10 dataset are performed using the same hyperparameters and network architecture as for the MNIST dataset. In Table 6.2 we can see the results of Contrastive Hebbian, its variants, the backpropagation baseline, and the Forward-Forward baseline. In this case, we do not have the results for the Soft-Hebb method as the test on this dataset was not performed on the original paper and no repositories with the code are provided. At first glance, we can notice a significant difference between the Forward-Forward and the backpropagation baseline, with FF being more effective. However, for this dataset, FF used receptive fields [39], which led to higher performance compared to backpropagation.

Table 6.2: Test set accuracy for the Contrastive Hebbian (CH) method in the CIFAR-10 dataset, its variants with label smoothing and without momentum, vanilla backpropagation and the Forward-Forward algorithm.

Method	**Accuracy (%)**	Standard deviation
CH	42.17%	± 0.38
CH + Label Smoothing	41.90%	± 0.32
CH without Momentum	38.89%	± 0.76
Backpropagation	**49.49**%	± 0.79
Forward-Forward[1] [14]	54.0%	-

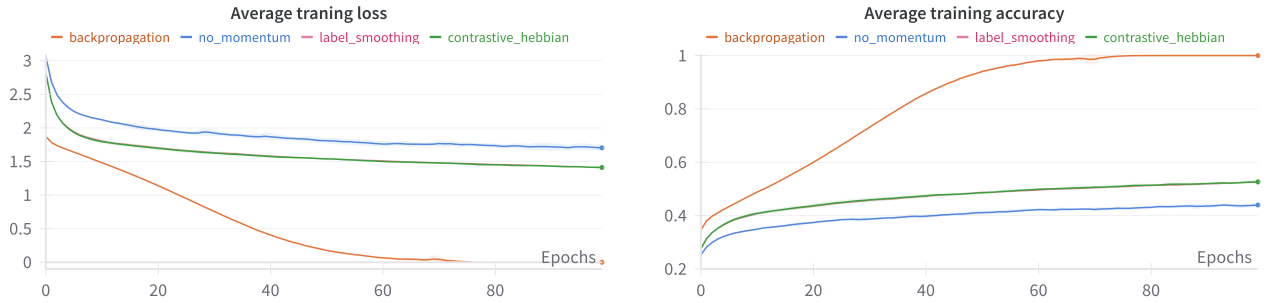


Figure 6.3: Average training loss and accuracy during the training phase over 5 independent runs on the CIFAR-10 dataset for linear networks. Note that the pink curve, relative to the label smoothing variant is not really visible because it is behind the curve of the Contrastive Hebbian original method.

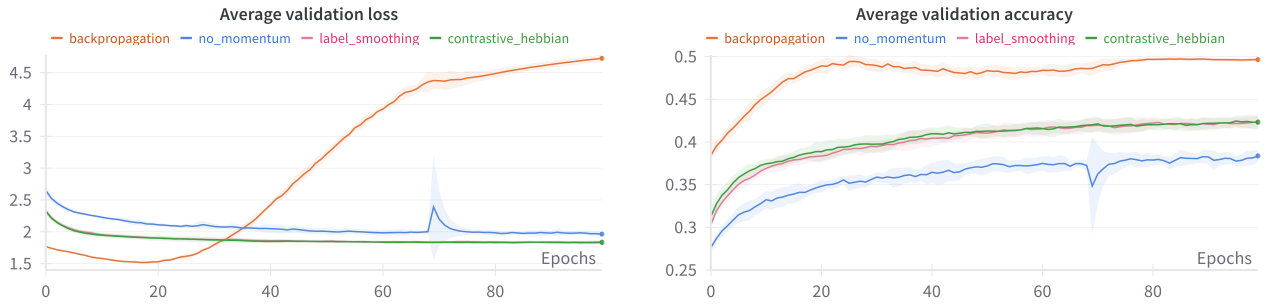


Figure 6.4: Average validation loss and accuracy during the training phase over 5 independent runs on the CIFAR-10 dataset for linear networks. Note that the pink curve, relative to the label smoothing variant is not really visible because it is behind the curve of the Contrastive Hebbian original method.

The performance of the Contrastive Hebbian method reaches an average accuracy of 42.17%, which is 7.38 points away from backpropagation. This is not a particularly high result; however, given the increased difficulty of the task compared to MNIST, we can consider this a satisfactory outcome. Considering that, again, the objective of this new rule is not to achieve state-of-the-art performance.

The variant with label smoothing again performs slightly worse than the vanilla algorithm, confirming the observation made for the previous tasks. The performance reduction when momentum is not employed is 3.28 points, higher than for the MNIST dataset.

In Fig. 6.3 and Fig. 6.4 we can see the training curves for the training and validation subsets, respectively. As we can see the network trained with classical backpropagation clearly overfits (orange

[1]The results reported in the paper "The Forward-Forward Algorithm: Some Preliminary Investigations" [14] for the CIFAR-10 dataset involves the use of receptive fields [39], so they are not really comparable

line) at a certain point during the training. The version using the momentum is well separated from the original algorithm while the variant with label smoothing is very close to it.

6.1.3 STL-10

For this dataset, the same architecture is employed, and all hyperparameters remain unchanged, except for the dropout rate, which is increased to 0.3 for all variants and backpropagation. This adjustment is motivated by the fact that STL-10 contains a very small amount of labeled data, making it more prone to overfitting. Another change is the learning rate for backpropagation, which is reduced to 0.0001. In this case, we do not have a comparable baseline reference; thus, the only baseline will be the standard backpropagation.

Table 6.3: Test set accuracy for the Contrastive Hebbian (CH) method on the STL-10 dataset, its variants with label smoothing and without momentum and vanilla backpropagation.

Method	**Accuracy (%)**	Standard deviation
CH	33.18%	± 1.18
CH + Label Smoothing	33.04%	± 0.61
CH without Momentum	31.83%	± 0.61
Backpropagation	37.01%	± 0.53

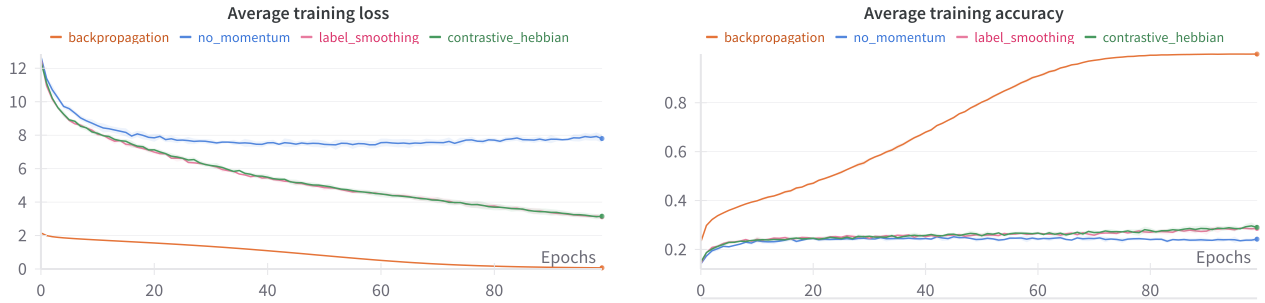


Figure 6.5: Average training loss and accuracy during the training phase over 5 independent runs on the STL-10 dataset for linear networks.

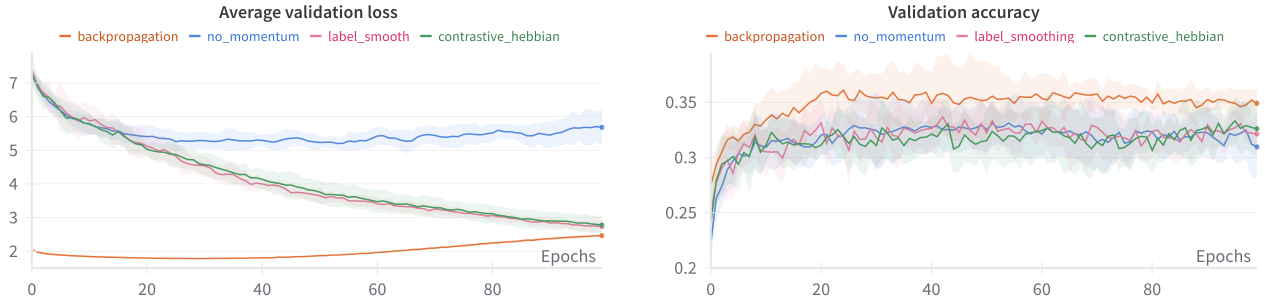


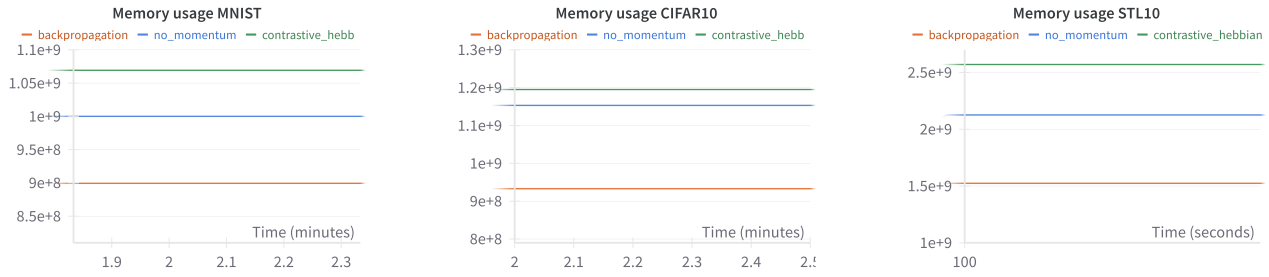
Figure 6.6: Average validation loss and accuracy during the training phase over 5 independent runs on the STL-10 dataset for linear networks.

From the results shown in Table 6.3 we can see that the proposed method performs only 4 percentage points lower than backpropagation, confirming that this approach can effectively learn complex representations in deep linear models. In Figures 6.5 and 6.6, the learning curves for all methods and variants are displayed. It is evident that backpropagation tends to overfit as training progresses,

while the contrastive Hebbian rule appears more resilient to this problem. Only the variant without momentum fails to converge to a solution: after 50 epochs, both the training and validation curves start to increase slightly, indicating that the distance to the optimal solution increases instead of decreasing. However, the same hyperparameters were used for all variants, and it is possible that the learning rate for this variant is too high. For the sake of coherence, these hyperparameters were not changed.

6.1.4 Memory Usage

In Fig. 4.3.2 the memory usage is shown for the Contrastive Hebbian method (green line), its variant without momentum (blue line), and backpropagation (orange line). As discussed in Section 4.3.2, both the variants with and without momentum should theoretically have the same memory requirements. However, this is not entirely true: the variant without momentum uses less memory than with momentum. This difference may arise from some optimization strategies employed by the `autograd` library at a lower level while managing the context. Note that backpropagation here uses SGD with momentum, which requires twice the number of parameters of the network in memory. This requirement is of the same order of magnitude as the proposed method. From the graphs, we can conclude that the observations made in Section 4.3.2 are correct only for the original variant using the momentum strategy. The higher memory usage of the Contrastive Hebbian algorithm is due to the vectors C_i and C_j and all the stored batches of data for each layer due to the double step employed. The difference in memory usage is more pronounced when the data size is larger: the dimensions of MNIST, CIFAR-10, and STL-10 are 1x28x28, 3x32x32, and 3x96x96, respectively. This is why the memory usage is proportionally higher for the STL-10 dataset.



(a) Memory usage on MNIST (b) Memory usage on CIFAR-10 (c) Memory usage on STL-10

Figure 6.7: Memory usage for the different variants and backpropagation, in this graph the number of bytes used is the metric on the y axis.

6.1.5 Conclusions

The performance of the Contrastive Hebbian framework and some of its variants is presented above. Notably, this method can be applied effectively to a deep linear network, overcoming the primary problem identified for all the rules analyzed in the literature review. Only the work of Moraitis et al. [34] proposed an effective multi-layer solution, but it was limited to two layers. Using label smoothing did not improve the performance of the algorithm as initially hypothesized in Section 4.1.3. Specifically, it reduced performance by less than 1 percentage point The variant without momentum appears to be effective, although there is a performance drop of around 2-3 percentage points. This variant, however, has a smaller memory requirement with respect to the version with momentum.

6.2 Experiments on a convolutional network

In this section, I will evaluate the proposed method using convolutional layers. The convolutional architecture used will be the same as the one proposed in "Hebbian Deep Learning Without Feedback" [18], shown in 6.10, and this will be the baseline to which I will compare my method. In Fig. 6.9 the details of the architecture used in my proposal are shown. Notably, it differs mainly in the removal of the BatchNorm layer and the activation function, the motivation of its removal is motivated in Section 4.3.4. In their work, Journé et al. [18] used a new custom and parametrizable activation function.

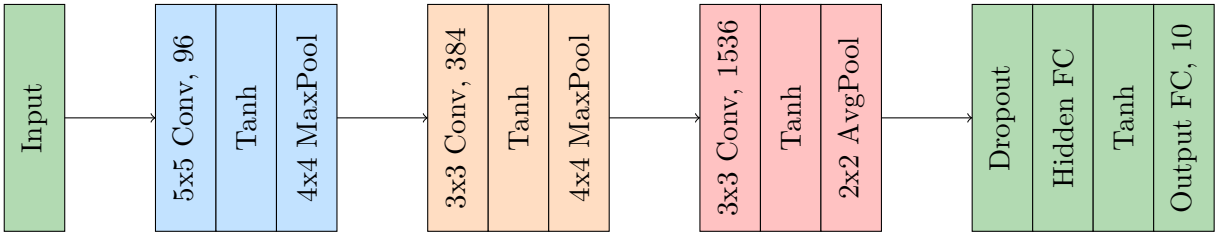


Figure 6.9: Architecture used by the model proposed in this thesis. In particular, the Hidden FC layer reduces the dimensionality of the output of the convolution by a factor of 4, to a maximum number of output values of 1024.

Moreover, two hidden layers are appended to the convolutional part, to show the efficacy at using the same rule for different types of layers.

To train in this scenario, an initial learning rate of 0.0001 is used for convolutional layers and 0.001 for linear ones, as in the previous section. The other hyperparameters remained unchanged with respect to the training on linear layers, and in particular, a learning rate of 0.03 and a weight decay of 0.003 are employed. In this case, instead of using an l2 normalization, weights are clipped to a range of [-10, 10] on the convolutional layer, linear layers still use l2 normalization as for the previous tests. The number of epochs used here is 50, aligned to the number of epochs used to train the SoftHebb linear head. The dropout rate used is 0.3 for the variant with a width scale of 4 and 0.1 with a scale factor of 2 and no dropout for a scale of 1, this is motivated by the fact that the bigger network is more prone to overfitting and thus a higher dropout rate is beneficial and vice versa; the dropout is applied only after the convolutional layers.

In this part, we will focus more on changing architectural parameters, in particular, I will analyze the performance decrease diminishing the convolutional scale factor, an important problem with all the Convolutional baselines that utilize Hebbian rules. As we can see in Fig. 6.8 only a factor of 4 is able to improve performance when depth increases: a hot point of the method proposed in this thesis will be to show that the decrease in performance will not be so high when diminishing that scale factor. Moreover, to achieve such results they tuned hyperparameters for each width factor with a greedy grid search. In my case, I will use always the same hyperparameters for all the variants. The variants using label smoothing and without momentum will not be analyzed in this section, as they do not lead to improved results. This exclusion is to avoid overcomplicating the complex comparison with the SoftHebb method. The order of analysis here is different: first, the results on CIFAR-10 are presented, and then on MNIST and STL-10. This is because the work "Hebbian Deep Learning Without Feedback" [18] focuses all the analysis on width scaling and different activation functions only on this dataset and thus MNIST and STL-10 will be less relevant in this chapter.

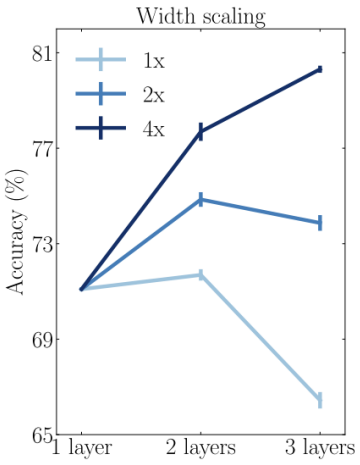


Figure 6.8: CIFAR-10 layer-wise performance of SoftHebb, for different width factors. Image taken from [18].

6.2.1 CIFAR-10

The results on the CIFAR-10 dataset are presented in this section. Table 6.4 and Table 6.5 contain the results on the test sets of all the different variants analyzed. Compared to the SoftHebb [18] method, the performance of my solution is not as competitive. However, the goal of my approach is not to achieve the best performance but to address the limitations discussed in Section 2.2.4. In Fig. 6.11 and Fig. 6.12 the training curves of all the proposed variants are shown. Note that in all the figures, the vanilla Contrastive Hebbian uses a scale factor of 4.

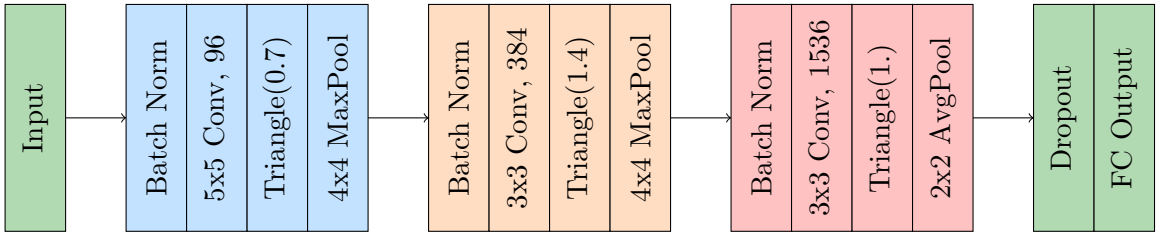


Figure 6.10: Architecture used by the SoftHebb convolutional model [18].

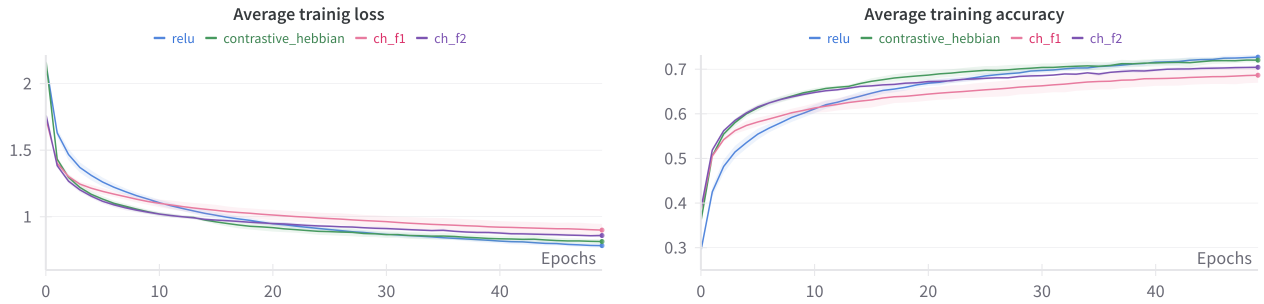


Figure 6.11: Average training loss and accuracy during the training phase over 5 independent runs on the CIFAR10 dataset for convolutional networks.

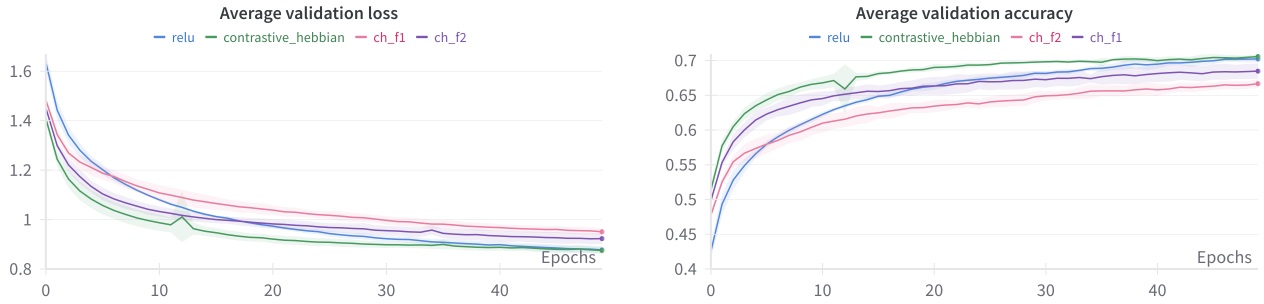


Figure 6.12: Average validation loss and accuracy during the training phase over 5 independent runs on the CIFAR-10 dataset for convolutional networks.

Different width scaling

The results of using different width scaling show that varying weight scales do not significantly compromise performance, unlike the SoftHebb method. In my case, the performance drop is around 2 percentage points for each decrease in width scale, whereas the other method shows a more significant drop, as illustrated in Fig. 6.8. Interestingly, when using a width scale factor of 1, the performance is comparable between the two methods.

The paper where SoftHebb was proposed [18] does not provide exact accuracy percentages but only the plot in Figure 6.8. Additionally, the code provided on GitLab[3] does not include the tuned

[1] This result comes from Fig. 6.8 and thus these are approximate values, but they are meaningful enough to give an idea on the behavior of this approach with different width scale factors

[2]This comes from the SoftHebb paper, the detail on how they trained such model like the optimizer used are not provided, however the difference in performance in respect to my training with backpropagation may be due to the misuse of the batch normalization in my method and the fact that I used an SGD with momentum and they may have been using an Adam optimizer

[3]https://github.com/NeuromorphicComputing/SoftHebb/blob/main/demo.py

Table 6.4: Test set accuracy for the Contrastive Hebbian (CH) with convolutions and SoftHebb methods on the CIFAR-10 dataset when different width scale factors are employed.

Method	**Accuracy (%)**	Standard deviation
CH, F-4	70.04%	± 0.24
CH, F-2	68.71%	± 0.76
CH, F-1	66.78%	± 0.52
SoftHebb, F-4 [18]	**80.31**%	± 0.14
SoftHebb, F-2 [18]	$\sim$ 75%[1]	-
SoftHebb, F-1 [18]	$\sim$ 66-67%[1]	-
Backpropagation	80.38%	± 0.52
Backpropagation [18]	84.00%[2]	-

parameters for different width scales other than 4. As cited in the paper, "where hyperparameters were tuned to each width factor" [18], these parameters are necessary to reproduce the same results statistically. However, it appears that the accuracy for a scale factor of 1 is around 66-67%, and for a scale factor of 2, it is around 74%. Given this observation, it is interesting to note that with a width factor of 1, the performance is comparable to SoftHebb, suggesting that my solution is more robust and could work well even with a smaller number of parameters.

Different activation functions

In their work, Journé et al. [18] state that the choice of the activation function is crucial for their method's performance. The performance with different activation functions is shown in Table 6.5. As observed, using standard activation functions drastically reduces the performance of their method, with ReLU being the best-performing non-parametrizable activation function, showing comparable performance to my approach. Interestingly, my proposed approach does not exhibit significant variation in performance when using different activation functions. In my case, I tested only tanh and ReLU to avoid using parametrizable functions and to keep the model less complex to train. Moreover, these two activation functions are the most common choices when training a new model or architecture. Softmax was not considered because it is almost never used in hidden layers as a pure activation function. It is typically utilized only in final layers to apply cross-entropy loss and in special architectures, such as the attention modules used in the Transformer architecture [45].

The method proposed in this thesis shows robustness on the change of activation function and in particular it seems that there is no decrease in performance at all using ReLU, in opposition to what was hypothesized in Section 4.3.3. In Fig. 6.12 we can see that the variant using ReLU takes more time to converge but at the end of the training it converges in the same way.

Table 6.5: Test set accuracy for the Contrastive Hebbian (CH) with convolutions and SoftHebb methods on the CIFAR-10 dataset when different activation functions are used. The width scale factor used here is 4 for all the experiments.

Method	**Accuracy (%)**	Standard deviation
CH, Tanh	70.04%	± 0.24
CH, ReLU	70.05%	± 0.44
SoftHebb, ReLU [18]	70.68%	± 1.07
SoftHebb, Tanh [18]	56.13%	± 0.34
SoftHebb, RePU [18]	79.08%	± 0.52
SoftHebb, Softmax [18]	54.05%	± 0.55
SoftHebb, RePU + Triangle [18]	**80.31**%	± 0.14

6.2.2 MNIST

The results on the MNIST dataset are presented in Tables 6.6 Here, only the results of the vanilla algorithms, specifically those with a width scale factor of 4, are compared with the work of Moraitis et al. [34], due to the lack of analysis on different variants in their work. Table 6.6 illustrates the minimal difference in performance for this task. However, it's important to note the simplicity of this task and the ease with which even naive methods can achieve high performance.

Table 6.6: Test set accuracy for Contrastive Hebbian (CH) with convolutions and SoftHebb methods on the MNIST dataset.

Method	**Accuracy (%)**	Standard deviation
CH	99.04%	± 0.09
SoftHebb [18]	99.35%	± 0.03
Backpropagation	99.42%	± 0.04

Method	**Accuracy (%)**	Standard deviation
CH, F-4	99.04%	± 0.09
CH, F-2	98.93%	± 0.10
CH, F-1	98.59%	± 0.15
CH, ReLU	99.05%	± 0.12

Table 6.7: Test set accuracy on different width scaling with Contrastive Hebbian (CH) and convolutions on MNIST dataset.

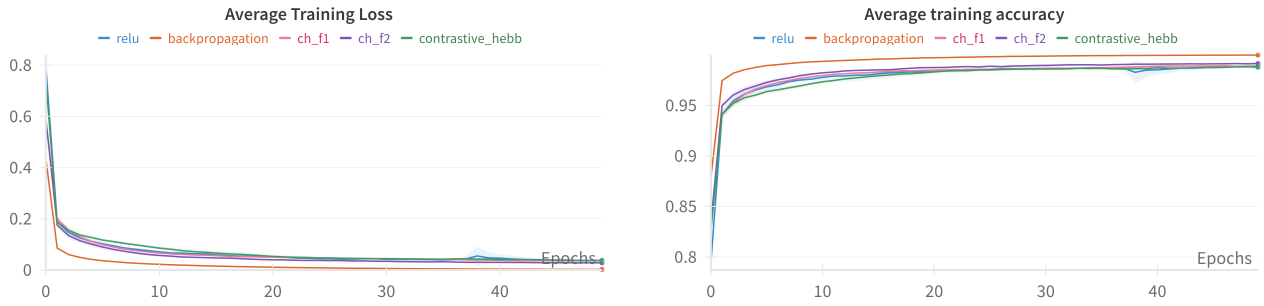


Figure 6.13: Average training loss and accuracy during the training phase over 5 independent runs on the MNIST dataset for convolutional networks.

The results for different activation functions and various scale factors are presented only for the proposed method in Table 6.7. Similar to the CIFAR-10 dataset, there is a very small decrease in performance when using a smaller width factor on convolutional layers. Using the ReLU activation function did not lead to any statistically significant difference in performance, however, it seems to show numerical instability in some cases. This suggests, again that the choice of the activation function is not so relevant and thus has one less hyperparameter to care about. Figures 6.13 and 6.14 display the training curves: as shown, during the training stage, the blue curve, relative to the variant using ReLU shows an important increase of the loss for one of the 5 runs. Moreover the curve relative to a width scaling of 1, in pink, is clearly distinguishable from the others and it shows its less effectiveness.

6.2.3 STL-10

The results on the STL-10 dataset are presented in this section: Table 6.8 contains the results on the test set for all the variants analyzed. In this case, the architecture changed and one more convolutional layer was added, aligning with what is done in the work of Journé et al. [18]. In particular, the new

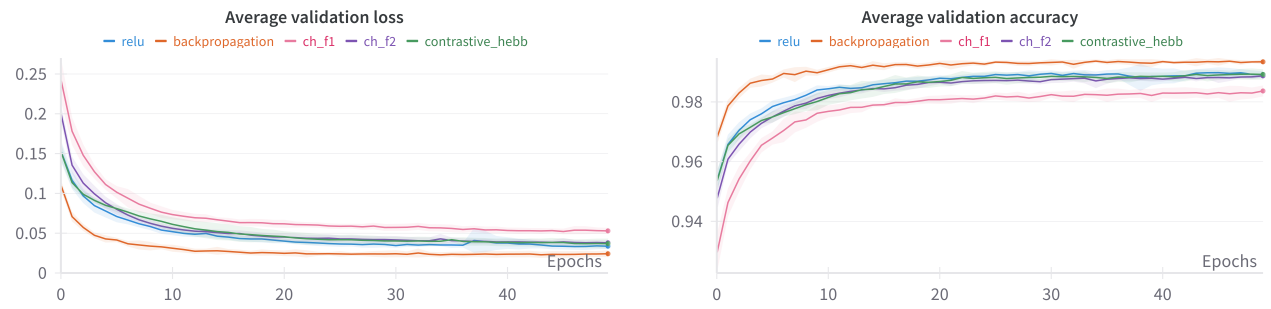


Figure 6.14: Average validation loss and accuracy during the training phase over 5 independent runs on the MNIST dataset for convolutional networks.

convolutional layer appended to the network has a 3x3 kernel with the average pooling only at the end of the convolutions. In this case, only the results on the vanilla algorithms, the ones with a width scale factor of 4, are shown because of the lack of analysis in the SoftHebb work. The results for different activation functions and different scale factors are presented only for the proposed method in Table 6.9

Table 6.8: Test set accuracy for Contrastive Hebbian (CH) with convolutions and SoftHebb methods on the STL-10 dataset.

Method	**Accuracy (%)**	Standard deviation
CH	60.55%	± 0.87
SoftHebb [18]	76.23%	± 0.19
Backpropagation [18]	74.51%	± 0.36

As we can see from Table 6.8 the Contrastive Hebbian method has a significantly worse performance than SoftHebb. This is due to the nature of the STL-10 dataset, which contains a very small amount of labeled data, increasing the difficulty of training models. While SoftHebb can utilize such unlabeled data, making it completely unsupervised, my method requires labeled data, making it impossible to use the unlabeled data for training. This is an important limitation, and further research should aim to develop different and unsupervised rules for this method. Notably, SoftHebb performs better than Backpropagation, again due to the limited amount of labeled data.

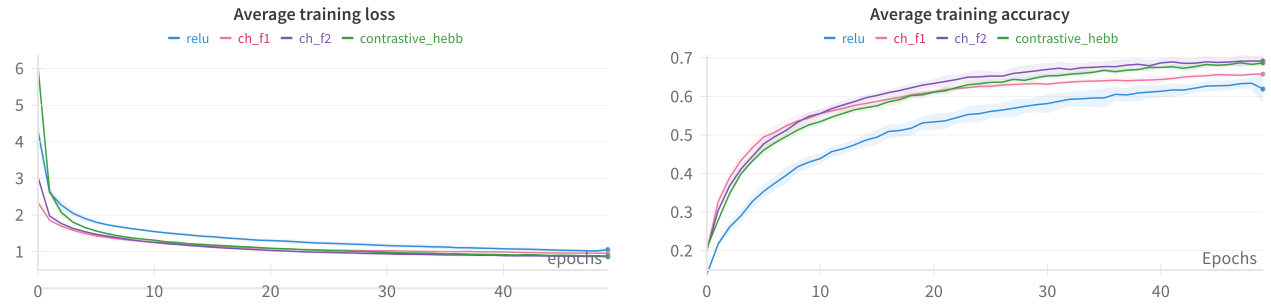


Figure 6.15: Average training loss and accuracy during the training phase over 5 independent runs on the STL-10 dataset for convolutional networks.

In Table 6.9, we can see the performance changes with different scale factors and the use of the ReLU activation function. Decreasing the scale factor from 4 to 2 does not worsen the performance significantly. However, with a width scaling of one, we lose 4 percentage points in accuracy. Another interesting observation is that using ReLU results in a 6-point drop in performance. This is somewhat

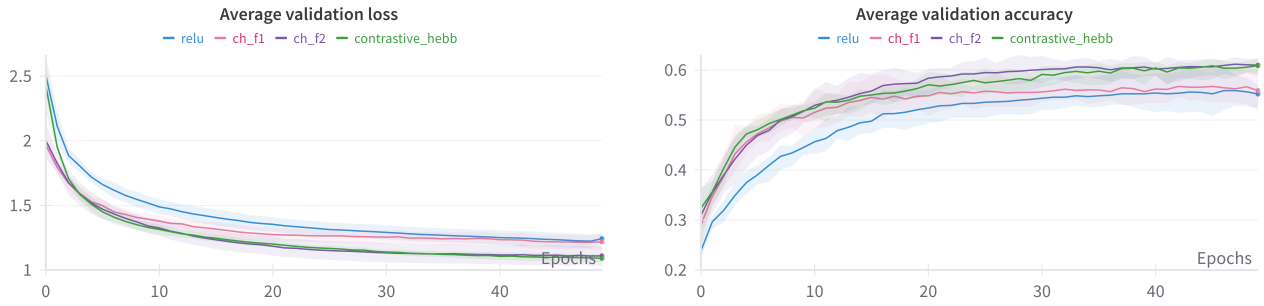


Figure 6.16: Average validation loss and accuracy during the training phase over 5 independent runs on the STL-10 dataset for convolutional networks.

in contrast to what was observed in the previous tasks where the choice of the activation function was not so important. However as observed in previous tests, this activation function with this method needs more time to converge and maybe 50 epochs are not enough to reach the same results.

Table 6.9: Test set accuracy on different width scaling on Contrastive Hebbian (CH) with convolutions on STL-10 dataset.

Method	**Accuracy (%)**	Standard deviation
CH, F-4	60.55%	± 0.87
CH, F-2	59.32%	± 0.84
CH, F-1	56.72%	± 0.24
CH, ReLU	54.49%	± 0.66

6.3 Continual Learning

In this section, I analyze the performance on the Split-MNIST task. Different hyperparameters are used for these experiments, with models trained for one epoch on each subtask: early stopping is an easy but powerful technique to avoid the network overfitting the presented data and thus having better generalization properties. Both methods utilize a width scale factor of 4 and a batch size of 64. The optimizer used is Stochastic Gradient Descent with momentum. Each variant is tuned to achieve the best performance on the tasks while maintaining a consistent architecture for both methodologies.

6.3.1 Split-MNIST

Table 6.10 presents the results for Split-MNIST. The following hyperparameters are employed for each method:

- **Contrastive Hebbian, Convolution:** A learning rate of 0.1 and a weight decay of 0.001 are used, with no dropout. The initial learning rate for the convolutional part is 0.0001 and 0.001 for the linear part. Normalization involves clipping values for convolutions and using L2 normalization for linear layers, coherently on what is done in the previous tests.
- **Contrastive Hebbian, Linear:** A learning rate of 0.01 and a weight decay of 0.003 are used, with a dropout rate of 0.3. The initial learning rate is 0.0001 with L2 normalization on the Hebbian rule.
- **Backprop Convolutional:** A learning rate of 0.0003 and a dropout rate of 0.1 are used.
- **Backprop Linear:** A learning rate of 0.0001 and a dropout rate of 0.1 are used.

Overall, the Contrastive Hebbian method with convolutions achieves the highest average accuracy of 84.8%, indicating a better resistance against catastrophic forgetting when using convolutions. For

Table 6.10: Results on the Split-MNIST task

Method	Task-1	Task-2	Task-3	Task-4	Task-5	Avg. Acc.
CH Conv.	80.4% ±6.1	88.5% ±2.6	68.1% ±2.5	97.2% ±1.0	89.7% ±1.6	**84.8**% ±1.7
CH Linear	87.0% ±3.2	84.5% ±1.3	64.3% ±3.6	95.2% ±1.6	73.5% ±1.9	80.9% ±1.1
BP Conv.	75.6% ±2.2	71.1% ±8.6	51.4% ±9.6	97.2% ±0.4	91.2% ±0.6	77.3% ±3.6
BP Linear	89.3% ±1.4	82.7% ±0.4	59.6% ±4.5	96.5% ±0.9	79.2% ±3.7	81.4% ±0.4

the linear layer part, both Contrastive Hebbian and Backpropagation show similar performance with an accuracy of 80.9% and 81.4%, respectively. This suggests that when using linear layers only, the same catastrophic forgetting issues are present. In particular, backpropagation shows worse performance when it comes to the use of convolutional networks with respect to its linear counterpart.

7 Conclusions

This thesis explored the intersection of Hebbian Learning, backpropagation, and contrastive learning, aiming to combine their strengths to develop a more biologically plausible and efficient deep learning algorithm. While Hebbian Learning offers theoretical advantages in terms of biological realism and potential for continual learning, previous attempts have struggled to scale to deeper networks while maintaining a simple and end-to-end training process. Experimental results on MNIST, CIFAR-10, and STL-10 datasets demonstrated the efficacy of the Contrastive Hebbian method.

7.1 Advantages

This work addressed several limitations of previous Hebbian Learning approaches. The key achievements of the proposed method are discussed in this section.

7.1.1 Extension to arbitrary deep network architectures

This is a significant contribution to the Hebbian Learning field. Prior works, such as "SoftHebb" [34] and "Hebbian Deep Learning Without Feedback" [18], allowed for effective Hebbian Learning in deep networks but were limited to either linear or convolutional layers, respectively. This thesis effectively demonstrates, for the first time, the adaptability of a Hebbian Learning method to arbitrary network architectures, including both linear and convolutional layers, as showcased in Section 6.2.

7.1.2 End-to-end training procedure

Another key improvement is the alignment of this framework with a standard end-to-end training paradigm. The methods mentioned above require non-standard training procedures, making their integration difficult into existing deep learning pipelines. The proposed Contrastive Hebbian framework does not break this paradigm, allowing for a more straightforward understanding and implementation. Moreover, unlike the Forward-Forward algorithm [14], which requires the creation of positive and negative samples and layer-wise training, the proposed method offers a simpler and more unified training process. Note that a simpler method does not mean that it is trivial, but conversely, it means that it will be easier for other researchers to take this work and integrate it into other frameworks or to use it as a baseline for finding better strategies or rules and further boost the research in this field.

7.1.3 Robustness against hyperparameters change

The work "Hebbian Deep Learning Without Feedback" [18] necessitates extensive hyperparameter tuning (as discussed in Section 2.2.4), and even slight modifications, such as changing the activation function or width scaling factor, can lead to drastic performance degradation. In contrast, the Contrastive Hebbian framework demonstrates significantly greater robustness. Experimental results show minimal performance drops with varying width scaling factors, and only slower convergence when using different activation functions. This robustness simplifies the training process and reduces the need for extensive hyperparameter optimization.

7.1.4 Substantial reduction in trainable parameters

The Neuron-Centric approach employed in this work, proposed by Ferigo et al. [9] and discussed in Section 2.1.7, drastically reduces the number of trainable parameters compared to Synaptic-Centric methods like those in Miconi's work on "Learning to Learn" framework [30].

7.1.5 Robustness against Catastrophic Forgetting

The Contrastive Hebbian method, when applied to convolutional networks, displays encouraging resistance to catastrophic forgetting. In experiments on the Split-MNIST task (Section 6.3), the Contrastive Hebbian method with convolutional layers achieved a higher average accuracy (84.8%) compared to backpropagation on the same architecture (77.3%). This suggests that the local, contrastive

updates promoted by the Contrastive Hebbian rule contribute to a more stable representation that is less susceptible to disruption when learning new tasks. This is not observed in a linear-only network where for both the modalities the results are similar: an average accuracy of 80.9% for the Contrastive Hebbian method and 81.4% for Backpropagation.

7.2 Disadvantages

Despite its advantages, the proposed Contrastive Hebbian method also has certain drawbacks that will be discussed in this section.

7.2.1 Memory requirements

While the Neuron-Centric approach [9] to Hebbian updates theoretically significantly reduces the number of trainable parameters compared to Synaptic-Centric methods, in practice the overall memory requirements during training remain quite high. As detailed in Section 4.3.2, the Contrastive Hebbian method necessitates storing several additional tensors during training. As illustrated in Figure 6.7, the Contrastive Hebbian method, even with the Neuron-Centric reduction, consumes more memory than standard backpropagation using SGD with momentum. The memory usage becomes even more pronounced for larger datasets like STL-10 due to the higher dimensionality of the input data. The reason here is the chain rule used by Backpropagation, which needs all the intermediate data for each operation that influences the trainable parameters. This increased memory requirement poses a practical limitation for training very large networks or for deploying the training algorithm on resource-constrained devices.

7.2.2 Performance

While the Contrastive Hebbian method achieves competitive performance on certain tasks, it is still not really competitive with state-of-the-art results achieved by methods like SoftHebb [18], particularly on challenging datasets like STL-10. Further research is needed to improve the performance of the method and close the gap with state-of-the-art approaches.

7.3 Future Directions

This research paves the way for several promising future directions:

- **Unsupervised Contrastive Hebbian Learning**: Exploring unsupervised or semi-supervised variants of the Contrastive Hebbian rule could leverage the abundance of unlabeled data and potentially lead to further performance enhancements.

- **Complete removal of Backpropagation**: Finding new strategies to adjust Neuron-Centric learning rates could allow one to get rid of the memory-consuming chain rule. Memory consumption is not the only problem: the backward pass to calculate the change in the learning rates slows the training speed. Getting rid of it could be very beneficial.

- **Testing on different task than classification**: In this thesis tests were performed only on classification tasks, however more dynamic and diverse tasks should be considered in order to assess the effectiveness of this method in a more diverse set of scenarios. It could be the case of pattern completion, reversal learning, and language modeling.

- **Deeper Convolutional Architectures**: Evaluating the method's performance on more complex convolutional architectures, such as those with residual connections or attention mechanisms, could provide insights into its scalability and capacity for learning even more sophisticated representations.

- **Adding pruning strategies**: Biological networks are inherently sparse, while artificial neural networks are dense, to make this approach more biologically plausible pruning strategies can be employed.

The Contrastive Hebbian method introduced in this thesis represents a significant advance in Hebbian Learning for deep neural networks. By combining contrastive learning principles, neuron-centric updates, and backpropagation, this work presents a promising direction toward biologically plausible, efficient, and continual learning in artificial neural networks. Future research building upon this foundation could lead to even more powerful and adaptable learning algorithms.

Bibliography

[1] Giuseppe Amato, Fabio Carrara, Fabrizio Falchi, Claudio Gennaro, and Gabriele Lagani. Hebbian learning meets deep convolutional neural networks. In Elisa Ricci, Samuel Rota Bulò, Cees Snoek, Oswald Lanz, Stefano Messelodi, and Nicu Sebe, editors, *Image Analysis and Processing – ICIAP 2019*, pages 324–334, Cham, 2019. Springer International Publishing. ISBN 978-3-030-30642-7.

[2] Suzanna Becker and Mark Plumbley. Unsupervised neural network learning procedures for feature extraction and classification. *Applied Intelligence*, 6(3):185–203, Jul 1996. ISSN 1573-7497. doi: 10.1007/BF00126625. URL https://doi.org/10.1007/BF00126625.

[3] Jonathan Binas, Ueli Rutishauser, Giacomo Indiveri, and Michael Pfeiffer. Learning and stabilization of winner-take-all dynamics through interacting excitatory and inhibitory plasticity. *Frontiers in Computational Neuroscience*, 8, 2014. ISSN 1662-5188. doi: 10.3389/fncom.2014.00068. URL https://www.frontiersin.org/articles/10.3389/fncom.2014.00068.

[4] Tom Binzegger, Rodney J. Douglas, and Kevan A. C. Martin. A quantitative map of the circuit of cat primary visual cortex. *Journal of Neuroscience*, 24(39):8441–8453, 2004. ISSN 0270-6474. doi: 10.1523/JNEUROSCI.1400-04.2004. URL https://www.jneurosci.org/content/24/39/8441.

[5] Ting Chen, Simon Kornblith, Mohammad Norouzi, and Geoffrey Hinton. A simple framework for contrastive learning of visual representations, 2020.

[6] Adam Coates, Andrew Ng, and Honglak Lee. An analysis of single-layer networks in unsupervised feature learning. In Geoffrey Gordon, David Dunson, and Miroslav Dudík, editors, *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, volume 15 of *Proceedings of Machine Learning Research*, pages 215–223, Fort Lauderdale, FL, USA, 11–13 Apr 2011. PMLR. URL https://proceedings.mlr.press/v15/coates11a.html.

[7] Francis Crick. The recent excitement about neural networks. *Nature*, 337(6203):129–132, Jan 1989. ISSN 1476-4687. doi: 10.1038/337129a0. URL https://doi.org/10.1038/337129a0.

[8] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(61):2121–2159, 2011. URL http://jmlr.org/papers/v12/duchi11a.html.

[9] Andrea Ferigo, Elia Cunegatti, and Giovanni Iacca. Neuron-centric hebbian learning, 2024.

[10] P. Földiák. Forming sparse representations by local anti-hebbian learning. *Biological Cybernetics*, 64(2):165–170, Dec 1990. ISSN 1432-0770. doi: 10.1007/BF02331346. URL https://doi.org/10.1007/BF02331346.

[11] Conrad C. Galland and Geoffrey E. Hinton. Deterministic boltzmann learning in networks with asymmetric connectivity. In David S. Touretzky, Jeffrey L. Elman, Terrence J. Sejnowski, and Geoffrey E. Hinton, editors, *Connectionist Models*, pages 3–9. Morgan Kaufmann, 1991. ISBN 978-1-4832-1448-1. doi: https://doi.org/10.1016/B978-1-4832-1448-1.50006-8. URL https://www.sciencedirect.com/science/article/pii/B9781483214481500068.

[12] Kaiming He, Haoqi Fan, Yuxin Wu, Saining Xie, and Ross Girshick. Momentum contrast for unsupervised visual representation learning, 2020.

[13] D. O. Hebb. The organization of behavior; a neuropsychological theory, 1949.

[14] Geoffrey Hinton. The forward-forward algorithm: Some preliminary investigations, 2022.

[15] Rasmus Kjær Høier and Christopher Zach. Two tales of single-phase contrastive hebbian learning, 2024.

[16] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift, 2015.

[17] Kazuki Irie, Róbert Csordás, and Jürgen Schmidhuber. Automating continual learning, 2024.

[18] Adrien Journé, Hector Garcia Rodriguez, Qinghai Guo, and Timoleon Moraitis. Hebbian deep learning without feedback, 2023.

[19] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2017.

[20] Alex Krizhevsky. Learning multiple layers of features from tiny images. Technical report, University of Toronto, 2009. URL `https://api.semanticscholar.org/CorpusID:18268744`.

[21] Dmitry Krotov and John J Hopfield. Dense associative memory for pattern recognition, 2016.

[22] Gabriele Lagani, Giuseppe Amato, Fabrizio Falchi, and Claudio Gennaro. Training convolutional neural networks with hebbian principal component analysis, 2020.

[23] Gabriele Lagani, Fabrizio Falchi, Claudio Gennaro, and Giuseppe Amato. Hebbian semi-supervised learning in a sample efficiency setting. *Neural Networks*, 143: 719–731, November 2021. ISSN 0893-6080. doi: 10.1016/j.neunet.2021.08.003. URL `http://dx.doi.org/10.1016/j.neunet.2021.08.003`.

[24] Brenden M Lake, Ruslan Salakhutdinov, and Joshua B Tenenbaum. Human-level concept learning through probabilistic program induction. *Science*, 350(6266):1332–1338, December 2015.

[25] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998. doi: 10.1109/5.726791.

[26] Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization, 2019.

[27] Wolfgang Maass. Networks of spiking neurons: The third generation of neural network models. *Neural Networks*, 10(9):1659–1671, 1997. ISSN 0893-6080. doi: https://doi.org/10.1016/S0893-6080(97)00011-7. URL `https://www.sciencedirect.com/science/article/pii/S0893608097000117`.

[28] Michael McCloskey and Neal J. Cohen. Catastrophic interference in connectionist networks: The sequential learning problem. *Psychology of Learning and Motivation*, 24:109–165, 1989. URL `https://api.semanticscholar.org/CorpusID:61019113`.

[29] Jan Melchior and Laurenz Wiskott. Hebbian-descent, 2019.

[30] Thomas Miconi. Learning to learn with backpropagation of hebbian plasticity, 2016.

[31] Thomas Miconi. Hebbian learning with gradients: Hebbian convolutional neural networks with modern deep learning frameworks, 2021.

[32] Thomas Miconi, Jeff Clune, and Kenneth O. Stanley. Differentiable plasticity: training plastic neural networks with backpropagation, 2018.

[33] Thomas Miconi, Aditya Rawal, Jeff Clune, and Kenneth O. Stanley. Backpropamine: training self-modifying neural networks with differentiable neuromodulated plasticity, 2020.

[34] Timoleon Moraitis, Dmitry Toichkin, Adrien Journé, Yansong Chua, and Qinghai Guo. Softhebb: Bayesian inference in unsupervised hebbian soft winner-take-all networks. *Neuromorphic Computing and Engineering*, 2(4), 2022. ISSN 2634-4386. doi: 10.1088/2634-4386/aca710. URL http://dx.doi.org/10.1088/2634-4386/aca710.

[35] Erkki Oja. Simplified neuron model as a principal component analyzer. *Journal of Mathematical Biology*, 15(3):267–273, Nov 1982. ISSN 1432-1416. doi: 10.1007/BF00275687. URL https://doi.org/10.1007/BF00275687.

[36] Roman Pogodin, Yash Mehta, Timothy P. Lillicrap, and Peter E. Latham. Towards biologically plausible convolutional networks, 2022.

[37] B.T. Polyak. Some methods of speeding up the convergence of iteration methods. *USSR Computational Mathematics and Mathematical Physics*, 4(5):1–17, 1964. ISSN 0041-5553. doi: https://doi.org/10.1016/0041-5553(64)90137-5. URL https://www.sciencedirect.com/science/article/pii/0041555364901375.

[38] Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, Gretchen Krueger, and Ilya Sutskever. Learning transferable visual models from natural language supervision, 2021.

[39] Rajesh P. N. Rao and Dana H. Ballard. Predictive coding in the visual cortex: a functional interpretation of some extra-classical receptive-field effects. *Nature Neuroscience*, 2(1):79–87, Jan 1999. ISSN 1546-1726. doi: 10.1038/4580. URL https://doi.org/10.1038/4580.

[40] Sashank J. Reddi, Satyen Kale, and Sanjiv Kumar. On the convergence of adam and beyond, 2019.

[41] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back-propagating errors. *Nature*, 323:533–536, 1986. URL https://api.semanticscholar.org/CorpusID:205001834.

[42] Andrea Soltoggio, Kenneth O. Stanley, and Sebastian Risi. Born to learn: The inspiration, progress, and future of evolved plastic artificial neural networks. *Neural Networks*, 108:48–67, December 2018. ISSN 0893-6080. doi: 10.1016/j.neunet.2018.07.013. URL http://dx.doi.org/10.1016/j.neunet.2018.07.013.

[43] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jonathon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision, 2015.

[44] Erik B. Terres-Escudero, Javier Del Ser, and Pablo García-Bringas. Emerging neohebbian dynamics in forward-forward learning: Implications for neuromorphic computing, 2024.

[45] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2023.

[46] Xiaohui Xie and H Sebastian Seung. Equivalence of backpropagation and contrastive hebbian learning in a layered network. *Neural Comput*, 15(2):441–454, February 2003.

Appendix A Code

A.0.1 Update rule for a hidden linear layer

```
    def update_rule_linear(self, target):
        if self.activation == RELU:
            out = F.relu(self.out)
        elif self.activation == TANH:
            out = torch.tanh(self.out)
        elif self.activation == SOFTMAX:
            out = self.out

        norms = torch.norm(out, dim=1, keepdim=True, p=2) + 1e-8
        out_norm = out / norms

        # Cosine similarity matrix
        similarity = out_norm @ out_norm.T # [batch, output_ch, output_ch]

        # Create masks based on class labels
        positive_mask = (target.unsqueeze(1) == target.unsqueeze(0)).float()
        negative_mask = 1.0 - positive_mask
        # set the diagonal to 0, negative_mask is already 0 there
        positive_mask.fill_diagonal_(0)

        # The loss aims to minimize the similarity between samples of the same class
        # and maximize the similarity between samples of different classes
        # loss = (positive_mask * (1 - similarity)
        #     + negative_mask * similarity.abs()).mean()

        # Add label smoothing to positive and negative masks
        if self.label_smoothing:
            positive_mask = (1 - self.label_smoothing) * positive_mask
                + self.label_smoothing / (self.out.shape[0] - 1)
            negative_mask = (1 - self.label_smoothing) * negative_mask
                + self.label_smoothing / (self.out.shape[0] - 1)
            positive_mask.fill_diagonal_(0)
            negative_mask.fill_diagonal_(0)


        # Compute the gradient of the loss w.r.t. similarity
        grad_similarity = negative_mask * similarity.sign() - positive_mask
        grad_similarity /= similarity.numel()

        # Compute gradient of similarity w.r.t. normalized output (out_norm)
        grad_out = grad_similarity @ out_norm + (grad_similarity.T @ out_norm)
```

```
        # Backpropagate through normalization
        grad_out = (grad_out * norms
            - out_norm * torch.sum(out * grad_out, dim=1, keepdim=True)) / norms**2
        # grad_out = grad_out_norm / norms ** 2

        # Backpropagate through tanh
        if self.activation == RELU:
            grad_out *= (out > 0).float().view_as(grad_out)
        elif self.activation == TANH:
            grad_out *= (1 - torch.tanh(out)**2).view_as(grad_out)
        elif self.activation == SOFTMAX:
            grad_out *= out

        # Compute the gradient of the loss w.r.t. the input
        grad_weight = grad_out.T @ self.x
        return - grad_weight
```

A.0.2 Update rule for a convolutional layer

```
    def update_rule_conv(self, target):
        # Reshape the output to [batch_size, output_ch * height * width]
        batch_size, out_channels, height, width = self.out.shape
        out = self.out.view(batch_size, -1) # [batch_size, height * width * output_ch]

        # the procedure to obtain the grad_out variable is the same so it is omitted
        [...]

        # reshape grad_out back to the shape of the convolutional layer's output
        grad_out = grad_out.view(batch_size, out_channels, height, width)

        # Compute the gradient with respect to the convolutional weights
        conv_weight_grad = torch.nn.grad.conv2d_weight(
            self.x,
            self.weight.shape,
            grad_out,
            self.stride,
            self.F_padding[0],
            self.dilation,
            self.groups
        )
        return - conv_weight_grad
```

A.0.3 Update rule for a final linear layer

```
    def target_update_rule(self, target):
        out = self.out.softmax(dim=1)
        return (target - out).T @ self.x
```