CITY UNIVERSITY LONDON

City University London
MSc Artificial Intelligence
Project Report
2024

Extending the Biological Plausibility and Efficacy of Hebbian Convolutional Neural Networks

Author: Julian Jimenez Nimmo
Student ID: 230066319
Supervisor: Dr Esther Mondragon
Submitted: 1^{st} October, 2024

By submitting this work, I declare that this work is entirely my own except those parts duly identified and referenced in my submission. It complies with any specified word limits and the requirements and regulations detailed in the assessment instructions and any other relevant programme and module documentation. In submitting this work I acknowledge that I have read and understood the regulations and code regarding academic misconduct, including that relating to plagiarism, as specified in the Programme Handbook. I also acknowledge that this work will be subject to a variety of checks for academic misconduct.

Signed: Julian Jimenez Nimmo

Abstract

This research explores the integration of Hebbian learning, a biologically plausible learning rule, into Convolutional Neural Networks (CNNs) for image processing tasks. Hebbian learning operates on local neural information, offering an alternative to the biologically implausible and computationally intensive backpropagation algorithm widely used in deep learning. Recent studies have combined Hebbian learning with current deep learning architectures, with distributed competition showing particular promise. Our project significantly extends the capabilities of Hebbian learning in CNNs, demonstrating performance that surpasses traditional backpropagation methods. A key finding is that Hebbian learning, when implemented with a hard competition mechanism, achieves state-of-the-art performance while exhibiting clear indications of hierarchical learning. This represents a crucial step towards more biologically realistic artificial neural networks. Furthermore, we introduce novel, biologically inspired competition mechanisms that enhance both the performance and generalisability of learnt representations. The research also includes architectural modifications inspired by both biological systems and current deep learning architectures. These changes result in networks with fewer parameters while maintaining high accuracy. Additional visualisation tool to understand Hebbian learning and its representations were also developed for interpretability and explainability.

Contents

1 Introduction and Objectives

Deep neural networks have achieved remarkable, often human-level, performance across diverse tasks in Artificial Intelligence (AI), including Computer Vision (CV), Natural Language Processing (NLP), and Reinforcement Learning (RL). These advancements, both in performance and architectural complexity, have been predominantly driven by the error backpropagation algorithm developed by Rumelhart et al. (1986), despite its biological implausibility.

Backpropagation, while effective, diverges significantly from biological learning mechanisms in several key aspects. Firstly, it assumes weights are symmetrical between forward and backward connections (synapses) between neurons, whereas biological synapses are unidirectional. Secondly, backpropagation updates weights globally after applying computations across all neurons through a backward pass, contrasting with the local, feedforward learning observed in biological neurons, which depends solely on the activity of locally connected neurons.

Current deep learning architectures trained with backpropagation face several practical challenges:

- **Resource Intensity:** They require substantial computational resources, including large volumes of labelled data (big data), high energy consumption, and extended training times, contributing to significant ecological impacts. Current Large-Language-Model (LLM) training has led to a rise in emissions from the leading AI companies, as stated by NPR Kerr (2024)
- **Limited Adaptability:** Despite proficiency in specific tasks, these models often struggle to adapt to new data or generalise effectively to unknown samples.
- **Vulnerability:** They exhibit fragility against adversarial attacks, introducing critical security vulnerabilities in deployed systems.

These limitations stand in stark contrast to biological intelligence, which demonstrates robustness to novel tasks and adversarial inputs, rapid learning from limited experiences, and remarkable energy efficiency.

Hebbian learning, a biologically inspired paradigm, attempts to bridge this gap by modelling neural plasticity through local learning rules. These rules update synaptic strengths based on the correlation between pre- and post-synaptic neural activities, more closely approximating biological learning processes.

1.1 Purpose, Product and Beneficiaries

The primary purpose of this project is to analyse and implement biologically-plausible learning rules from the Hebbian learning family and biologically-inspired components within a deep learning architecture. We aim to evaluate features learnt by these models and compare their performance against traditional backpropagation methods.

This research will involve implementing state-of-the-art (SOTA) Hebbian learning algorithms to identify key factors enabling effective learning, analyse differences between various Hebbian methods and propose and test improvements to narrow the performance gap with backpropagation.

The project will yield the following products:

- Creation of versatile neural network models capable of training via backpropagation, biologically inspired learning rules, or hybrid approaches.
- Updated, computationally optimised implementation of state-of-the-art biologically plausible learning algorithms.

- Novel learning algorithms and inter-neuronal competition mechanisms for training neural network architectures.
- Comprehensive comparative analysis of Hebbian and backpropagation learning in terms of performance, efficiency, and biological plausibility.

This research will benefit several stakeholder groups, with a focus on **AI Researchers** exploring alternative learning paradigms for continuous learning and computational efficiency, **Neuroscientists and Biologists** employing artificial neural networks to model neuronal behaviour in the brain, **Industry Practitioners** requiring models capable of continuous learning from unlabelled data or models with low energy consumption and rapid training times, and **Theoretical Neuroscientists** investigating the computational principles underlying biological learning to bridge the gap between artificial and biological neural networks.

1.2 Research Questions

The primary question driving this research project is:

Can Hebbian learning algorithms achieve representational learning comparable to backpropagation while maintaining biological plausibility?

Additionally, we will explore the following sub-questions:

- How do different Hebbian learning rules affect the network's ability to form hierarchical representations?
- What role does neuronal competition play in Hebbian learning?

1.3 Objectives

To address the research questions, we have established the following objectives:

- Develop a baseline model implementing a simple Hebbian learning rule:
  - Utilise datasets and parameters consistent with existing research.
- Implement and validate state-of-the-art Hebbian approaches, ensuring replication of reported performance, learning rules, and competition modes.
- Identify limitations in current state-of-the-art methods and propose extensions, focusing on both optimisation and performance enhancements.
- Conduct comparative analyses between Hebbian and backpropagation learning across multiple metrics through visualisations of filters for interpretability and other metrics:
  - Classification accuracy.
  - Learning speed and computational efficiency.
  - Learnt representations and differences in synaptic connections

1.4 Methods

We will employ Python 3. as the primary programming language, with PyTorch serving as the deep learning framework and GitHub as our version control. Additional libraries include:

- Matplotlib and Seaborn for data visualisation
- Scikit-learn and Umap for data manipulation and evaluation metrics
- NumPy for numerical computations

- Weights and Biases (WanDB) for real-time training visualisation and analysis

To accomplish our objectives, the following methodologies were implemented:

1. CIFAR-10 Dataset with ZCA Whitening
2. Hard-WTA Hebbian CNN implementation similar to Miconi (2021), Lagani, Falchi, Gennaro & Amato (2022) implementations.
3. SoftHebb CNN implementation by Journé et al. (2022)
4. Visualisation tools to understand and interpret neural learning
5. Pre-Synaptic competition
6. Temporal Competition and Statistical Competition modes
7. BCM learning rule
8. Lateral Inhibition through Surround Modulation Kernel
9. Depthwise Separable Convolution Hebbian Architecture
10. Residual Hebbian Block
11. Hebbian Network following Dale's Principle

The implementation for this project can be found on our GitHub repository:
`https://github.com/Julian-JN/Master_Thesis_Hebbian_CNN`

To ensure comparable experimental setups and learning rules, we will reference and adapt code from the following publicly available projects:

- HebbDemo: `https://github.com/GabrieleLagani/hebbdemo`
- Lagani Thesis Research: `https://github.com/GabrieleLagani/HebbianLearningThesis`
- SoftHebb: `https://github.com/NeuromorphicComputing/SoftHebb`

1.5 Structure of Report

The report is structured as follows:

- Section 2: Context
  - Introduction of a neuron in biological and artificial terms
  - Explanation of backpropagation and its biological implausibility
  - Overview of a Convolutional Neural Network architecture, its biological counterpart and its mathematical foundation
  - Biological evidence for Depthwise Separable Convolutional architecture
  - Overview and description of biologically-plausible Hebbian learning rule and its variants
  - Analysis of neuronal competition methods enabling specialisation
  - Survey of recent techniques, implementations and results in the field
- Section 3: Methods
  - Neural network architecture design considerations

  - Dataset processing techniques
  - Detailed description of implementation decisions for baseline, state-of-the-art, and novel extensions
  - Design of a comprehensive visualisation board to analyse learnt features
  - Elaboration on different configurations
  - Detailed explanation of the experimental setup
- Section 4: Results
  - Comprehensive presentation of results from various configurations
  - Quantitative and qualitative analysis of results
  - Visualisation of learned features, activation patterns, and performance metrics
- Section 5: Discussion
  - Evaluation of results with respect to project objectives.
  - Discussion of new findings and answer to research questions
- Section 6: Conclusion
  - Reflection on key findings and their implications for the field.
  - Evaluation of limitations and potential avenues for future work.
  - Concluding remarks on the viability of Hebbian learning as an alternative to backpropagation and overview of the project.

This structure ensures a comprehensive treatment of the research topic, from theoretical foundations to practical implementation and analysis of results.

2 Context

2.1 Neuron Model

A neuron, in both biological and artificial contexts, functions as a fundamental unit of information processing and transmission (Lagani et al. 2023). In neuroscience, it refers to a specialised cell that processes and transmits information through electrical and chemical signals. In the field of Artificial Intelligence (AI) and Artificial Neural Networks (ANNs), a neuron is an abstracted mathematical model that mimics the basic function of its biological counterpart.

The learning process in both biological systems and ANNs involves modifying the strength of connections between neurons. In biology, this is achieved through synaptic plasticity, while in ANNs it's implemented through algorithms like backpropagation.

Neurons can form deep networks by organising into layers, with each consecutive layer processing increasingly abstract representations of the input data. In deep neural networks, multiple hidden layers between the input and output layers allow for the modelling of complex, hierarchical relationships in data, mimicking the layered structure observed in biological neural systems.

A simplified biological neuron model can be modelled artificially as presented in Figure 1, where synaptic connections or weights are described by a vector w. The neuron takes as input a vector x, and produces an output $y = f(\sum x^T w)$ where f is an activation function. In biological and neuroscience terms, input values x are also called pre-synaptic activations, while the output y is termed post-synaptic activation.

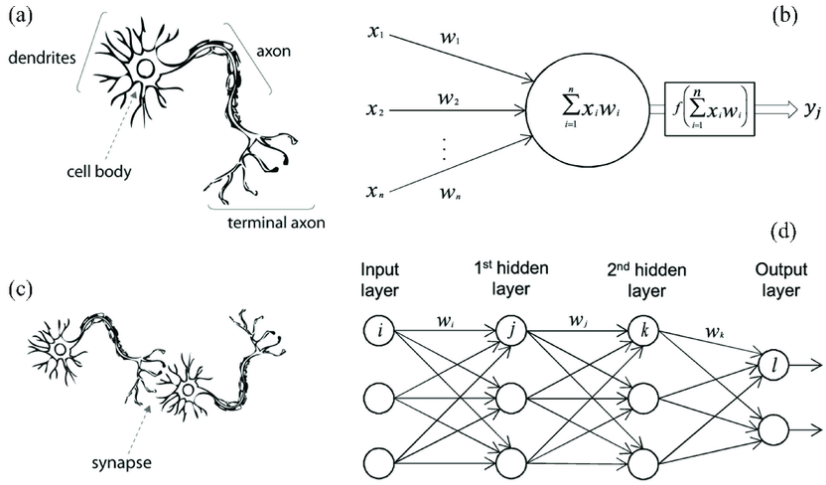

Figure 1: Comparison of biological and artificial neural models and networks (Churhe 2024)

Biological neurons operate under a multitude of physiological constraints (Pulvermüller et al. 2021), among which several principles are of importance in understanding their functionality. The principle of **locality** states that neurons are capable of direct interaction solely with their immediate neighbours via synaptic connections, constraining the scope of connectivity. A property of these synaptic connections is their capacity for **synaptic plasticity**, where the strength of synaptic connections can be modulated over time in response to activity.

Dale's Principle states that presynaptic neurons undergo either an exclusively excitatory or inhibitory influence on their postsynaptic counterparts,d. Furthermore, the propagation of information to neighbouring neurons in the network is restricted to excitatory synapses, establishing a unidirectional flow of excitatory signals in neural circuits (Shepherd & Grillner 2018).

ANNs trained with backpropagation fully respect plasticity, as it can increment or decrease the strength of a connection. The locality constraint in backpropagation is only maintained during a feedforward pass, as synaptic updates utilise global information from non-local neurons. Dale's Principle and the propagation constraint are typically not respected, as a neuron in backpropagation networks can have a mixture of excitatory and inhibitory connections to different neighbours, which has been proven to aid with representation learning.

However, new research by Cornford et al. (2020) using a backpropagation network which respects Dale's Principle and propagates information using excitatory synapses, with explicit excitatory neurons and inhibitory inter-neuron cells in a feedforward-inhibition architecture, has achieved comparable state-of-the-art performance.

2.2 Discrete-Time Weight Update and Backpropagation

In the context of ANNs, a learning rule is an algorithm that governs how the network adapts and modifies its synaptic connections in response to input data. The primary goal of a learning rule is to enable the network to improve its performance on a given task over time by learning from experience.

The discrete time learning rule is the theoretical algorithm behind ANNs. This rule operates on the principle that the network's state and weight updates occur at discrete time steps, rather than continuously across time. Current Von Neumann hardware, such as CPU and GPU, use discrete electrical signals while biological neurons operate with continuous electrical signals.

A general form for a discrete-time synaptic plasticity rule can be expressed as the change in a connection between neuron i and j, commonly referred to as synaptic strength or weight w, at time step t, based on a function f over inputs $x(t)$ and synaptic strengths $w(t)_{ij}$:

$$\Delta w(t)_{ij} = f(x(t), w(t)) \tag{1}$$

and the update to the weight as:

$$w(t+1)_{ij} = w(t)_{ij} + \Delta w(t)_{ij} \tag{2}$$

While the input $x(t)$ can be considered as the output of neuron i serving as input for neuron j, and could be expressed as $x(t)_i$, we adopt the simpler notation $x(t)$ for clarity. The time-dependent notation for weights and other terms in the equation can be simplified as we assume a discrete-time weight modification. For instance, $w(t)$ is represented as w, omitting the explicit time dependence. These notations simplification are adopted for all terms, unless not implicit.

A batch formulation of this general learning rule can be developed as follows, where multiple inputs are presented at the corresponding time step:

$$\Delta w = f(x^{(1)}, ..., x^{(N)}, w) \tag{3}$$

All learning rules in this project work in discrete time steps given its focus on current hardware implementations, although extensions of these rules can be applied to continuous time scenarios, such as Hebbian extensions with Spike-Timing-Dependent-Plasticity (STDP), as performed in a CNN by Kheradpisheh et al. (2018). However, the backpropagation training paradigm behind the success of deep learning operate strictly in discrete time, given its theoretical and mathematical formulation.

2.2.1 Theoretical Foundations of Backpropagation Learning

The learning process in deep neural networks is driven by the efficient backpropagation algorithm (Rumelhart et al. 1986) and the gradient descent optimisation rule, which can be expressed in a simplified form as:

$$\Delta w = -\eta \nabla_w E(x, w) \tag{4}$$

where η represents the learning rate, E denotes the calculated loss function, and ∇_w symbolises the gradients associated with parameters w. This equation encapsulates how neural networks learn from data, but to understand how it modifies its weights we must define the concepts of loss functions, gradients and gradient descent, and backpropagation.

2.2.2 Loss Functions and Error Attribution

The loss function serves as a crucial metric in quantifying the discrepancy between a neural network's predictions and the actual target values or true labels, guiding the learning process (Rojas & Rojas 1996). The choice of loss function is context-dependent, varying based on the nature of the problem at hand as it decides how the task is optimised.

For regression tasks, the Mean Squared Error (MSE) is a common choice:

$$E_{MSE} = \frac{1}{N} \sum_{i=1}^{N} (y_i - \hat{y}_i)^2 \tag{5}$$

Here, y_i represents the true value and $\hat{y}_i$ the predicted value. The quadratic nature of MSE ensures that larger errors are penalised more heavily, driving the model to minimise significant deviations.

In classification scenarios, the Cross-Entropy loss, also known as Softmax loss, is typically employed:

$$E_{CE} = -\sum_{i=1}^{C} y_i \log(\hat{y}_i) \tag{6}$$

where C denotes the number of classes, y_i is the true probability of class i, and $\hat{y}_i$ is the predicted probability. This loss function is particularly effective in multi-class problems, as it encourages the model to output probability distributions that match the true class distributions.

The loss function plays an essential role in assigning contributions to the error for each parameter in the network. By computing partial derivatives of the loss with respect to each parameter through gradients, we can determine how much each weight or parameter contributes to the overall error. This attribution of error is the key to backpropagation's success, also known as **Credit Assignment** (Bengio & Frasconi 1993), as it allows the network to understand which parameters need adjustment and to what degree.

2.2.3 Gradients and Learning Dynamics

Gradients represent the rate of change of the loss function with respect to each parameter in the network (Amari 1993). Mathematically, for each weight w_i, the gradient is defined as:

$$\frac{\partial E}{\partial w_i} = \lim_{\epsilon \to 0} \frac{E(w_i + \epsilon) - E(w_i)}{\epsilon} \tag{7}$$

This partial derivative indicates the inclination and magnitude of the steepest increase in the loss landscape for a given parameter. The negative of this gradient points in the direction of steepest descent, which is the direction we move to minimise the loss, as visualised in Figure 2.

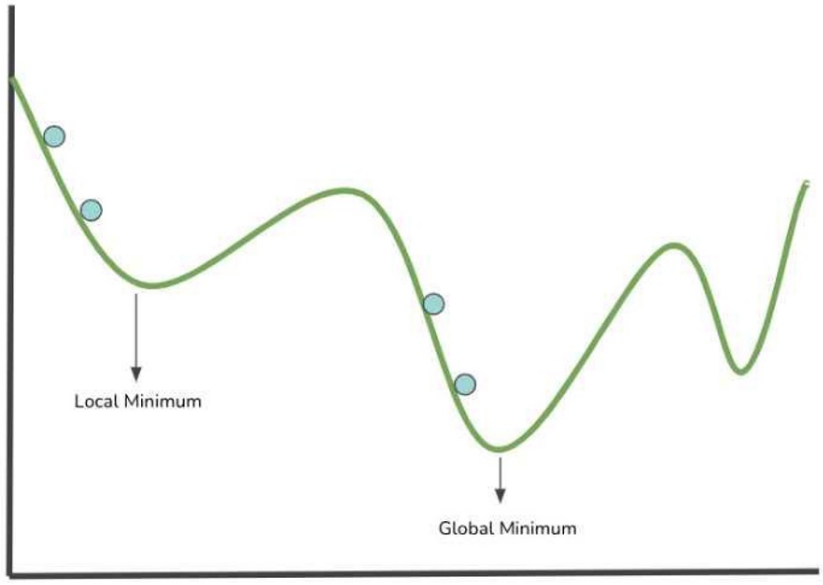

Figure 2: Simplified diagram of a loss landscape, where the aim of the backpropagation is to minimise this loss using gradients (Kucharlapati 2024)

Moving in the opposite direction of the gradient refers to the process of adjusting the network's parameters to reduce the loss. This is the foundation behind learning in current neural networks. By iteratively making small adjustments in the direction opposite to the gradient, the network gradually improves its performance, minimising the discrepancy between its predictions and the target values. This iterative process is known as **Gradient Descent**.

The computed gradients are directly used to update the weights of the network. A basic Gradient Descent weight update rule can be expressed as:

$$w_{new} = w_{old} - \eta \frac{\partial E}{\partial w} \tag{8}$$

This equation illustrates how each weight is adjusted in proportion to its contribution to the error, as determined by the gradient. The learning rate η in the gradient descent equation controls the size of these steps. A larger learning rate may lead to faster convergence but risks finding a local minimum, while a smaller learning rate ensures more stable learning but requires more iterations to reach the optimal point.

2.2.4 Backpropagation Algorithm

Backpropagation is an efficient algorithm for computing gradients in deep neural networks (Amari 1993), (Rumelhart et al. 1986). The term *propagating the error backwards* refers to the process of calculating how much each neuron in the network contributed to the overall error, starting from the output layer and moving towards the input layer.

This backward flow of information allows the network to adjust its weights and reduce the error. The process applies the chain rule of calculus to efficiently compute gradients for all parameters in the network.

For a neuron j in layer l, the gradient is computed as:

$$\frac{\partial E}{\partial w_{ji}^l} = \frac{\partial E}{\partial a_j^l} \frac{\partial a_j^l}{\partial z_j^l} \frac{\partial z_j^l}{\partial w_{ji}^l} \tag{9}$$

where a_j^l is the activation of neuron j in layer l, z_j^l is its weighted input, and w_{ji}^l is the weight of the connection from neuron i in layer $l-1$ to neuron j in layer l.

This equation demonstrates how the error signal is decomposed and attributed to individual weights in the network. By applying this computation recursively through the layers, backpropagation efficiently calculates the gradients for all parameters, enabling the network to learn from its mistakes with gradient descent and incrementally improve its performance.

2.2.5 Advanced Optimisation Techniques

While the interplay between loss functions, gradients, and backpropagation forms the core of learning in neural networks, various algorithms have been developed to enhance convergence (Ruder 2016) and overcome challenges such as local minima. Optimizers improve upon basic gradient descent by addressing issues such as varying gradient magnitudes in different dimensions and the need for adaptive learning rates:

Stochastic Gradient Descent (SGD) updates weights using a single or small batch of samples, introducing beneficial noise that can help escape local minima. Momentum accelerates SGD by accumulating a velocity vector in directions of persistent reduction in the objective across iterations, helping to overcome areas of shallow gradients.

More sophisticated algorithms like Adam (Adaptive Moment Estimation) adapt the learning rate for each parameter, combining ideas from momentum-based methods and utilising moving averages of past gradients. These adaptive methods can lead to faster convergence and improved performance. A general form encompassing many of these optimizers can be expressed as:

$$w_{t+1} = w_t - \eta_t m_t \tag{10}$$

where η_t is the learning rate at time t, and m_t is a function of past gradients, specific to each optimizer.

2.2.6 Biological Implausibilities

Despite its effectiveness in ANNs, backpropagation faces several significant biological implausibilities that limit its direct application to neuromorphic hardware (Campbell et al. 2022). These issues highlight the divergence between current machine learning techniques and our understanding of biological learning processes (Lillicrap et al. 2020).

Gradient descent, as implemented in backpropagation, requires non-local error signals calculated from neurons that are not directly connected in the network (Song et al. 2020). This process, known as **Non-Local Plasticity**, involves propagating information backwards through the network, utilising information from neurons at different time steps and network locations. Such non-local information exchange is not observed in biological neural systems, where synaptic plasticity is primarily influenced by local interactions between pre- and post-synaptic neurons.

The reliance on non-local plasticity introduces computational inefficiencies in ANNs, as both forward and backward pass variables must be stored in memory. This requirement significantly increases the energy and resource demands of the learning process.

Another biological implausibility from backpropagation is the **Weight Transport** issue. The problem arises from backpropagation's requirement to use the transpose of the weight matrix

during the backward pass to update gradients. This requires the calculation of the transposed matrix for every neuron and its connections, further increasing computational demands. Biological neurons utilise unidirectional synapses, and the assumption of bidirectional connections with symmetric weights implied by weight transport lacks empirical support in neuroscience.

In backpropagation, the error for a neuron can only be computed after a complete forward and backward pass through the entire network. This phenomenon, known as **Update Locking**, impedes rapid learning as the network cannot process new samples until the bidirectional propagation is completed for the current sample. Update locking is biologically implausible because neurons in biological systems do not exhibit such precise coordination in modifying their synapses.

Backpropagation relies on a differentiable **Global Loss** function that provides a top-down supervision signal. This approach frames learning as a task of specialisation through loss minimisation. It can lead to issues such as overfitting, increased sensitivity to adversarial attacks, and limited generalisability. Biological learning processes are believed to involve more localised and distributed forms of error correction.

These biological implausibilities alongside its discrete timing constraint prevent the direct implementation of backpropagation-based learning rules in energy-efficient, brain-inspired neuromorphic hardware (Schuman et al. 2022). Such hardware is typically designed for local rule-based training paradigms with continuous signals that more closely align with our current understanding of biological neural plasticity. The discrepancies between backpropagation and biological learning mechanisms have spurred research into alternative learning algorithms that maintain the computational power of backpropagation while adhering more closely to biologically plausible principles.

2.3 Convolutional Neural Networks

Convolutional Neural Networks (CNN) are a deep neural network architecture designed for image or multi-dimensional data processing (LeCun et al. 1998), typically trained using the backpropagation training algorithm described in Section 2.2. A simple form of this architecture is presented in Figure 3.

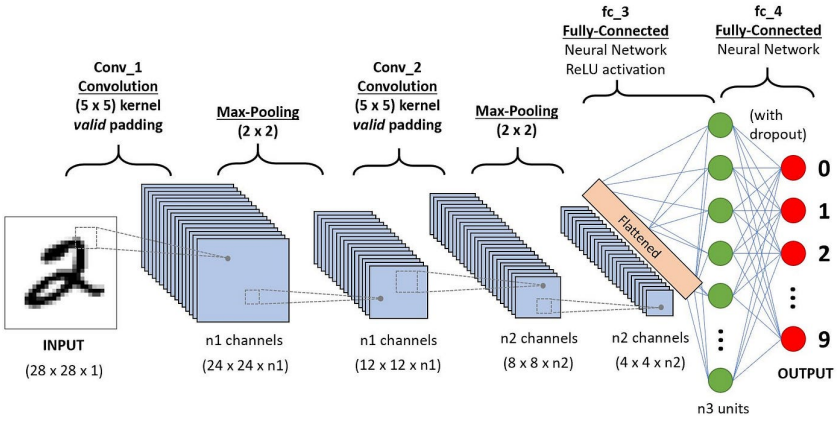

Figure 3: Overview of the CNN architecture (Mayur 2024)

2.3.1 Biologically Inspired Architecture

This convolutional architecture (Fukushima 1980) is inspired by the visual cortex in the brain, mimicking the processing of visual senses in biological systems. The primary visual cortex (V1)

found in mammals is organised in a manner which allows different neurons to specialise in detecting distinct features in small regions of the visual field known as receptive fields.

Neurons in the visual cortex are organised in a hierarchical structure where simple cells in the first regions respond to specific orientations of edges and colours, while more complex cells respond to more complex features by integrating information from multiple simple cells.

CNNs are designed to mimic this hierarchical processing, with convolutional layers simulating neurons in the visual cortex in different regions. Pooling layers aggregate the information from multiple neurons, preserving the most informative features while reducing the spatial resolution. For the hierarchical structure, CNNs use multiple layers to build up from simple patterns to more complex representations. This mimicry is illustrated in Figure 4.

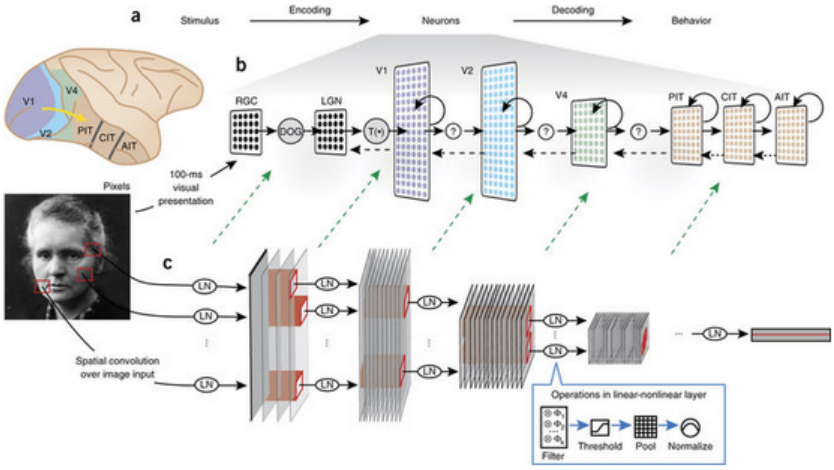

Figure 4: Comparisons between visual cortex and CNN (Lindsay 2018).

In CNNs, neurons are computational units that perform part of the convolution operation and have two key properties: each neuron is connected only to a small region of the input volume, known as its receptive field; and neurons in the same depth/channel share the same weights, implementing a key CNN concept called **weight sharing**.

Weight sharing is a fundamental concept in CNNs that ties together the ideas of neuron weights and convolutional filters. It significantly reduces the number of parameters in the network and provides translation invariance. In a given convolutional layer, multiple neurons (typically arranged in a 2D grid) share the same set of weights. This shared set of weights is equivalent to a single convolutional filter.

Each convolutional filter can be thought of as a single neuron that is replicated across the entire input space. The weights of the replicated neuron are aggregated together during an update for shared weights. The output of applying a single filter across the entire input creates a feature map. All neurons in this feature map share the same weights (filter), but operate on different spatial locations of the input.

A convolutional layer typically consists of multiple filters, each producing its own feature map. Neurons across different feature maps have different weights, allowing the network to detect various features. Mathematically, for any neuron in a particular feature map, its output can be represented as:

$$y_{i,j} = f(W * x_{i,j} + b) \tag{11}$$

where W is the shared weight matrix, $*$ denotes the convolution operation, $x_{i,j}$ is the input patch, b is a shared bias term, and f is the activation function.

2.3.2 Theoretical and Mathematical Definitions

To understand CNNs more intuitively (Wu 2017), the components of this architecture are broken down below:

Convolutional Layers: These can be thought of as feature detectors. Each convolutional filter slides over the input, looking for specific patterns (like edges, textures, or more complex shapes in deeper layers). In a mathematical notation, convolutional layers can be defined as below, where a filter/kernel K is applied to the input X:

$$Y_{i,j,c} = \sum_{m=1}^{k} \sum_{n=1}^{k} \sum_{c'=1}^{C_{\text{in}}} X_{i+m-1,j+n-1,c'} \cdot K_{m,n,c',c} \tag{12}$$

This 3D input has shape has shape $H \times W \times C_{in}$ where H is height, W is width and C_{in} is the number of input channels. The kernel is a 4D tensor of shape $k \times k \times \times C_{in} \times C_{out}$ where k is the spatial dimensions of the filter, C_{in} is the number of input channels and C_{out} the number of output channels. The output Y will have size $H' \times W' \times C_{out}$ where H' and W' are determined by stride s and padding p. The result of the convolutional operation will have output shape defined by:

$$\text{Output size} = \left(\frac{H - k + 2p}{s} + 1 \right) \times \left(\frac{W - k + 2p}{s} + 1 \right) \times C_{\text{out}} \tag{13}$$

After each convolutional layer, an activation function is typically applied element-wise to introduce non-linearity. The Rectified Linear Unit (ReLU) is a common choice as it helps the network learn complex patterns by allowing for non-linear decision boundaries.:

$$f(x) = \max(0, x) \tag{14}$$

Pooling Layers: These layers perform down-sampling. They help the network focus on the most important information by reducing the spatial dimensions. The pooling layer consists of the following pooling operation:

$$Z_{i,j,c} = \max_{m,n \in \{1,\ldots,k\}} Y_{i+m-1,j+n-1,c} \tag{15}$$

where $Z_{i,j,c}$ is the output of the pooling layer at position (i, j) at channel c. The output size of a pooling layer is derived from:

$$\text{Output size} = \left(\frac{H' - k}{s} + 1 \right) \times \left(\frac{W' - k}{s} + 1 \right) \times C_{\text{out}} \tag{16}$$

While max pooling is common, other pooling methods exist. **Average Pooling** takes the average value in each pooling window, while **Global Pooling** applies pooling across the entire spatial dimensions and is often used before fully connected layers.

Fully Connected Layers: These layers take the high-level features learnt by the convolutional and pooling layers and use them to make the final classification or prediction. The fully connected

layer uses flattened convolutional outputs (1D vector) f as the input:

$$o = W \cdot f + b \tag{17}$$

where w is the weight matrix and b is the bias vector.

Modern CNN architectures often incorporate additional elements. **Residual Connections** allow training for very deep networks by providing skip connections between layers (He et al. 2016), **Inception Modules** use multiple filter sizes in parallel to capture features at different scales (Szegedy et al. 2015) and **Depthwise Separable Convolutions** reduce computational cost by separating spatial and cross-channel correlations (Chollet 2017).

These advancements have led to state-of-the-art performance in various computer vision tasks, including image classification, object detection, and semantic segmentation.

2.4 Depthwise Separable Convolutions

Depthwise Separable Convolutions, developed by (Chollet 2017), is an architectural modification of a standard CNN which divides convolutional operations into two steps, seen in Figure 5: a depthwise convolution followed by a pointwise convolution. This architectural change reduces the number of parameters and computations required while maintaining similar performance and reducing overfitting.

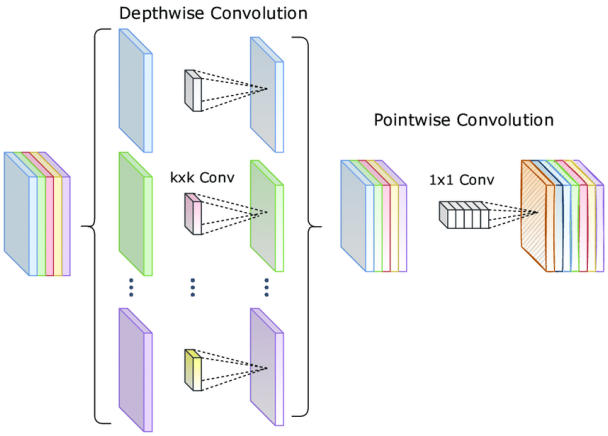

Figure 5: Depthwise Separable Convolution architecture (Sultonov et al. 2022).

Depthwise convolution step applies a single convolutional filter per input channel. If the input has M channels, it uses M filters. This step captures spatial relationships within each input channel separately and independently.

Pointwise convolution applies a convolution of dimensions 1x1, which means it operates on a single pixel (or feature) at a time across all input channels. They take an input with dimensions (H, W, M) and produce an output with dimensions (H, W, N). This step creates new features by computing linear combinations across channels.

Depthwise separable convolutions are considered more biologically plausible than regular convolutions. The separation of spatial (depthwise) and feature combination (pointwise) operations aligns with the specialisation observed in biological neurons. They create sparser connections between layers, which is more similar to biological neural networks than the dense connections in regular convolutions. The two-step process mimics the hierarchical nature of feature extraction in the visual cortex, where lower levels process basic spatial features, and higher levels combine these features.

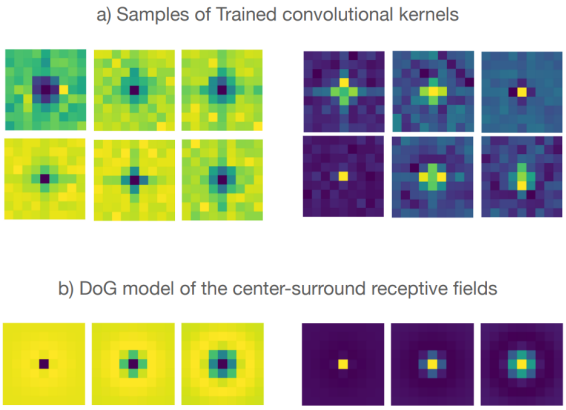

Figure 6: Centre-surround kernels which emerge in trained depthwise networks (Babaiee et al. 2024)

The depthwise step processes each channel independently, which is analogous to how different types of visual information (e.g., colour, orientation) are processed in parallel pathways in the early visual system. The pointwise step allows for the integration of information across different channels, similar to how neurons in higher cortical areas combine various types of visual information.

Recent research by Babaiee et al. (2024) has found the kernels of backpropagation trained depthwise separable convolutional networks exhibit centre surround on- and off-centre kernels, as seen in Figure 6. These kernels can be seen in biological receptive fields, marking a biological link between this convolution architecture and the visual cortex. They extended this discovery with an initialisation scheme to facilitate learning of these receptive fields.

2.5 Hebbian Learning Rules

Hebbian learning is a biologically-inspired alternative to backpropagation. Weights in synapses increase with respect to the activities in pre- and postsynaptic stimuli at a given time step (Lagani et al. 2023). This ability to modify its strength is called *synaptic plasticity*, which should be able to increase through *Long-Term Potentiation* or LTP, or decrease through *Long-Term Depression* or LTD.

2.5.1 Conceptual Overview of Hebbian Learning

At its core, Hebbian learning embodies the idea that "neurons that fire together, wire together." This principle suggests that the connection between two neurons strengthens when they activate simultaneously. In the context of artificial neural networks, this translates to adjusting the weights between neurons based on their correlated activity.

Key aspects of Hebbian learning include updates depend only on the activity of locally connected neurons, not on global network performance. No external "teacher" or error signal is required, allowing for **Unsupervised Learning**. The rule captures statistical correlations in the input data, and can provide a mechanism for storing and retrieving patterns producing **Associative Learning** (Magee & Johnston 1997).

These learning rules can be applied to discrete time Artificial Neural Networks (ANN), although biological neurons work in continuous time. The basic **Hebbian** Learning rule as described by Hebb can be expressed as:

$$\Delta w(t) = \eta y(x, w)x \tag{18}$$

where $y(x, w)$ or y is the post-synaptic activation of the neuron, which is a function of the input and the weights. It strictly follows the principle of *cells that fire together wire together*.

This learning rule applied to each individual neuron, and requires only local connections between neurons. This learning rule assumes weights are non-negative. While Equation 18 captures the essence of associative learning, it suffers from unbounded weights which can only grow in strength, leading to instability in neural network dynamics. This limitation has motivated the development of modified Hebbian rules.

2.5.2 Variants of Hebbian Learning

A modification to regulate unbounded growth is the **Grossberg Instar** learning rule (Grossberg 1976), which introduces a decay term proportional to the value of the weight and regulated by the value of the activation. Small weight modifications occur when the output is low, and vice versa.

$$\Delta w(t) = \eta y(x - w) = \eta(yx - yw) \tag{19}$$

When $\eta y(x, w) \leq 1$, this learning rule obtains the geometric interpretation of weights converging towards clusters of inputs, with each step acting proportionally to the similarity between input and weight. If a similar input is presented, the neuron will evoke a stronger response to it, and repeated presentation of inputs will allows convergence to the centroid of the input distribution.

This learning rule belongs to the family of *post-synaptic gating* rules, as the postsynaptic activity controls the extent of the update. In this form of Hebbian learning rules, the generic form can be expressed as:

$$\Delta w(t) = \eta y(x - \theta) \tag{20}$$

where the threshold θ determines how updates occur. This formulation allows for bidirectional synaptic plasticity, a key feature observed in biological neural systems. LTP is induced when the pre-synaptic activity is weaker than θ, otherwise LTD will occur.

Conversely *pre-synaptic gating* rules inverts the contribution of x and y. The threshold in these rules can be fixed on the weights, on current activations or on history of activations.

$$\Delta w(t) = \eta x(y - \theta) \tag{21}$$

The **Covariance rule** (Sejnowski & Tesauro 1989) combines both pre- and postsynaptic influences by using running averages of both pre- and post-synaptic activities, based on the biological concept of synaptic traces.

$$\Delta w(t) = \eta(y - \theta_y)(x - \theta_x) \tag{22}$$

It more accurately reflects the temporal dynamics of synaptic plasticity and incorporates the concept of metaplasticity, where the history of synaptic activity influences future plasticity.

The Bienenstock-Cooper-Munro **BCM** (Bienenstock et al. 1982) is a variation of the pre-synaptic learning rules, and introduces nonlinearity function ψ in the learning:

$$\Delta w(t) = \eta x \psi(y - \theta) \tag{23}$$

where θ_y and θ_x represent thresholds or running averages of post- and pre-synaptic activities. This rule also introduces metaplasticity and temporal dynamics to the synaptic modifications.

This threshold is not fixed, as this would lead to unstable updates as in the basic Hebbian rule. By allowing a dynamic threshold, learning can be stabilised. This threshold must grow faster than the postsynaptic activity as this activity can grow to large values. As such, non-linearity $\psi(y - \theta)$ is defined as $y(y - \theta)$, where θ is the moving average of the postsynaptic activities

A different suggestion to stabilise updates is the **Oja's** learning rule (Oja 1982), which normalises the weight vector at each update, keeping vector lengths constant but allowing directions to change. This allows the weights to perform online Principal Component Analysis (PCA) (Becker & Plumbley 1996).

$$\Delta w(t) = \eta y_i (x - \sum_{j=1}^{i} y_j w_j) \tag{24}$$

This learning rule does add some biological implausibilities, as it requires knowledge of weights and postsynaptic activities of other neurons, which may not be directly available in biological systems.

2.6 Competition between Neurons

As neural networks are comprised of multiple interconnected neurons, it is important to have different neurons encode unique information. Competition mechanisms allow neurons to specialise and avoid convergence to similar weights as other neurons. These mechanisms play a crucial role in self-organising systems and contributes to the formation of efficient neural representations.

2.6.1 Winner-Takes-All (WTA) Competition

Winner-Takes-All or WTA is a hard competition (Rumelhart & Zipser 1985), where a singular winner is allowed to update their weights. It allows neurons to perform clustering on a set of data points and promotes a sparse representation of the data. The competition can be based on the maximum activation between neurons, or the similarity between input and weight vectors, using angular or euclidean distances.

Mathematically, the WTA mechanism can be formalised as follows:

$$i^* = \arg\max_i y_i \tag{25}$$

where i^* is the index of the winning neuron and y_i is the activation of neuron i.

This competition can be included for different spatial locations between within-layer neurons, locally connected neurons, or across the whole network. It can also be used across other dimensions, such as between neurons/filters of different channels in CNN, which is the predominant choice in Hebbian-CNN implementations Amato et al. (2019). This hard competition can also be modified to allow Top-K winner selection.

WTA competition induces a form of sparse coding, where only a small subset of neurons are active for any given input. This sparsity has been shown to have several advantages, improving

energy efficiency in neural computation, enhancing pattern separability to facilitate subsequent learning in classification tasks and Increasing storage capacity in associative memory models.

This competition can be included in any Hebbian rule, by substituting the postsynaptic activity y for a competition coefficient r, which can be expressed in different ways:

$$\Delta w(t) = \eta r(x - w) \tag{26}$$

where r can be expressed as either a binary mask denoting which neurons won the competition, or as a mask over the post-synaptic activities. This coefficient r will express the reward given to the neurons in different competition modes:

$$\text{binaryMask}_i = \begin{cases} 1 & \text{if } i = \arg\max_j y_j \\ 0 & \text{otherwise} \end{cases} \tag{27}$$

where y_j is the activation of neuron j. Now, we can express r_i in two forms:

$$r_i = \text{binaryMask}_i \tag{28}$$

$$r_i = y_i \cdot \text{binaryMask}_i \tag{29}$$

Equation 28 represents a hard Winner-Takes-All mechanism, where only the winning neuron (with highest activation) receives a non-zero r_i. Equation 29 represents an activity-masked Winner-Takes-All, where the winning neuron's r_i is set to its activation y_i, while all other neurons use $r_i = 0$.

2.6.2 Soft Winner-Takes-All Competition

A softer form of this competition was suggested (Nowlan 1989), as Hard-WTA enforces a quantised representation of the data as singular neurons respond when a specific pattern occurs in the data. Neural networks, both biological and trained on backpropagation, exhibit a distributed representation, where multiple neurons encode different patterns of the data conjointly.

Soft-WTA applies a Softmax function over the activations, allowing all neurons to learn in a distributed manner. Neurons with higher activation functions receive a higher score. A temperature parameter T can be included to control the variance of the activations.

$$\text{softmax}(y) = r = \frac{e^{y_i/T}}{\sum_{j=1}^{n} e^{y_j/T}} \tag{30}$$

As $T \to 0$, the Soft-WTA approaches the Hard-WTA, while as $T \to \infty$, it approaches uniform competition. The temperature parameter T provides an interpolation between these extremes. Similarly, this can be used directly instead of the postsynaptic activity or used to mask the postsynaptic activities.

2.7 Anti-Hebbian and Lateral Inhibition

Other forms of competition can be included to promote diversity between neurons, such as Anti-Hebbian Learning and Lateral Inhibition:

2.7.1 Anti-Hebbian Learning

Anti-Hebbian learning consists of the negation of the Hebbian update rules (Choe 2022), where synaptic strengths between neurons is weakened if they are activated together. The purpose of learning the opposite of Hebbian learning is to decorrelate activities between neurons. It has biological evidence in the CA1 region of the hippocampus, where certain interneurons exhibit Anti-Hebbian plasticity and contribute to the temporal coding of information

Mathematically, a basic Anti-Hebbian rule can be expressed as:

$$\Delta w_{ij} = -\eta x_i y_j \tag{31}$$

where Δw_{ij} is the change in synaptic weight from neuron j to neuron i, η is the learning rate, and x_i and y_j are the activations of neurons i and j respectively.

It induces competition by discouraging co-activation of similar neurons, while increasing diversity in the learnt representations by decorrelating neurons. This ensures each neuron learns a distinct pattern, as well as reducing the noise from data to help separate sensory data processing.

Hebbian and Anti-Hebbian learning was incorporated as a loss function for backpropagation networks in HaH by Cekic et al. (2022). Anti-Hebbian learning can be applied to all non-winners of a competition mode. SoftHebb applies this idea to all neurons expect the maximally activated neuron which updates using Hebbian updates, helping decorrelate activities while using a distributed learning rule.

2.7.2 Lateral Inhibition

Lateral Inhibition is a related concept, which focuses on inhibiting the activity of neurons based on the activity of neighbouring neurons (Gabbott & Somogyi 1986). Similarly, this approach enhances the contrast and competition between neurons. It is achieved by preserving the most informative neurons active, while inhibition suppresses the activity of redundant neurons. This enhances sparsity in the representations.

Lateral inhibition is present in biological neural systems, particularly in sensory processing circuits. One of the most well-studied examples of lateral inhibition occurs in the retina. Horizontal cells provide inhibitory feedback to photoreceptors and bipolar cells.

In computational neuroscience and machine learning, lateral inhibition serves to suppress weakly activated neurons, and enhances the signal-to-noise ratio in neural representations. In visual processing, lateral inhibition can enhance edge detection and other low-level feature extraction processes. By promoting competition between neurons, lateral inhibition naturally leads to sparse neural codes, which have been shown to have computational and representational advantages.

This can be applied to neural networks through inhibitory connections which use Anti-Hebbian learning rules. WTA can be considered an extreme form of lateral inhibition, where neighbouring neurons are completely suppressed. Lateral anti-Hebbian connections between neurons in a CNN architecture were used by Pogodin et al. (2021) to showcase weight sharing can be accomplished in a locally-connected CNN. Using a sleep-like phase which uses Hebbian plasticity, it provided a local and biological solution to the weight sharing principle in neural networks.

Lateral inhibition was implemented by Hasani et al. (2019) as a convolutional operation, with a kernel replicating the surround modulation in biological visual systems. During modulation, each neuron excites nearby neighbours and inhibits far away neighbours within a range based on the level of its own activity. It was found to increase feature saliency and decrease redundancies while reducing training steps necessary for convergence.

2.8 Unsupervised and Hybrid Hebbian-CNN Research

Hebbian learning, an intrinsically unsupervised process, operates without the need for external feedback or supervisory signals to guide the learning trajectory. This learning rule facilitates online learning through continuous and incremental updates to synaptic strengths as each sample is presented.

To quantify the efficacy of this learning process, features extracted via unsupervised layers are typically passed through a single classifier layer. The MNIST (LeCun et al. 1998) and CIFAR-10 (Krizhevsky et al. 2009) datasets serve as standard benchmarks in this field, with backpropagation methodologies achieving over 99% on both datasets, thereby establishing a rigorous baseline for comparative analysis.

2.8.1 Unsupervised and Hybrid Hard-WTA

Recent investigations by Amato et al. (2019) have explored the application of Hebbian learning to Convolutional Neural Networks (CNNs) in an unsupervised context , yielding promising outcomes through diverse architectural configurations and competitive mechanisms. Lagani et al. achieved notable success, with 98.55% accuracy on MNIST and 65% on CIFAR-10 utilising a hard-WTA framework with 3 convolutional layers.

Their methodology employed cosine similarity as both the competitive criterion and activation function. Crucially, data whitening emerged as an essential pre-processing step for the enhanced performance with this hard competition. The researchers systematically evaluated architectures comprising up to six layers, observing a consistent degradation in performance with each additional layer.

Hybrid approaches integrating layers trained via backpropagation with those employing Hebbian learning have yielded intriguing insights (Amato et al. 2019). Empirical evidence suggests that the use of Hebbian learning in both initial and final layers can be executed without compromising performance. However, the integration of Hebbian learning within intermediate layers was associated with a decline in efficacy.

These results indicate that hard-WTA competition is suitable for shallow network architectures and for the fine-tuning of lower and higher layers in pre-trained networks. Notably, this approach achieves convergence in fewer epochs relative to backpropagation while reducing computational resource requirements. Nevertheless, its efficacy in training intermediate layers is limited, and performance decreases with depth, contrasting with the performance gained in backpropagation as depth increases.

2.8.2 Unsupervised Gradients Hard-WTA

Miconi (2021) integrated Hebbian learning and gradient-based networks, incorporating layer-wise loss functions equivalent to various Hebbian learning rules. This approach utilised hard-WTA competition based on post-synaptic activations alongside data whitening. Additional implementations of synaptic pruning, triangular activation functions, and adaptive thresholds to maintain homogeneous firing rates among neurons facilitated the propagation of sparse representations across layers, culminating in a 65% accuracy on CIFAR-10.

These findings set in motion the search for distributed Hebbian representations, as the localised representations in individual neurons might cause the efficiency decrements in deep networks and impede the formation of hierarchical representations.

2.8.3 Unsupervised Hebbian-PCA

Lagani, Falchi, Gennaro & Amato (2022) explored the Sanger/Oja learning rule without competition, termed HPCA (Hebbian Principal Component Analysis), to promote distributed coding capable of extracting principal components from input data. Employing the same experimental framework to their hard-WTA research but removing the requirement for data whitening, HPCA yielded comparable results and advantages. Significantly, it exhibited reduced performance degradation in deeper network configurations, maintaining accuracies around 60% in a six-layer network architecture.

2.8.4 Unsupervised SoftHebb

The SoftHebb model, introduced by Journé et al. (2022), incorporates soft-WTA competition as its foundational framework. This competitive mechanism, in conjunction with a novel plasticity learning rule, has demonstrated the capacity to minimise cross-entropy loss without explicit access to labels under specific conditions. The model employs softmax over post-synaptic activations for competitive selection and implements anti-Hebbian learning on all neurons except the maximally activated unit. The rule can be described below:

$$\Delta w_{ik}^{(SoftHebb)} = \eta \cdot y_k \cdot (x_i - u_k \cdot w_{ik}). \tag{32}$$

where w_{ik} is a synaptic weight from a pre-synaptic neuron i with activation x_i, y_k is the postsynaptic output of a neuron k, and u_k is the result of the post-synaptic softmax competition.

A key innovation is the introduction of per-neuron adaptive learning rates derived from weight vector norms, substantially accelerating learning such that it requires a single epoch. To achieve high performance, convolutional width-scaling where each successive layer increases in width by a factor of four, facilitates the formation of hierarchical representations across multiple layers. These advancements enabled the network to achieve 80% accuracy on CIFAR-10, representing the current state-of-the-art performance among unsupervised Hebbian approaches.

2.8.5 Hebbian Resilience

Comparative analysis of these Hebbian methodologies extend beyond performance metrics to encompass robustness evaluations (Gupta et al. 2022). When compared with backpropagation, Hebbian learning exhibits several notable advantages: convergence rates up to 20 times faster, performance improvements of up to 20% in limited data and computational resources scenarios, and enhanced resilience against both noise and adversarial attacks.

2.9 Literature Suggested Improvements

To improve the efficacy of these Hebbian methodologies, literature has suggested the extension of these local rules to new biologically-inspired convolutional architectures, with inter-layer backward feedback, top-down feedback connections from higher to lower layers, and recurrent connections.

Recurrent and top-down feedback has improved performance for models trained with backpropagation in object recognition tasks (Spoerer et al. 2017), reducing the error in scenarios with occlusion of objects and providing robustness against noise. A recurrent neural network trained on local reward-modulated Hebbian learning rule (Miconi 2017) showcases similar neural activity present in higher visual cortex, offering a plausible architectural model of the animal cortex.

Further improvements suggested by Lagani (2024) for bio-inspired deep learning include pre-synaptic competition, which introduces competition to the inputs x of a layer, BCM learning

rule implementation in a Hebbian-CNN, new competition modes based on statistical information of the data or temporal history of activations, and Information-theory based approaches.

In WTA competition especially in the convolutional setting, adjacent patches of an image can lead to neurons learning translated and correlated filters. Mechanisms which allow spatial decorrelation allow for richer and distributed spatial representations.

3 Methods

This chapter details the key implementation considerations that guided the development of our biologically plausible neural network model. The design philosophy encompasses several crucial aspects aimed at creating a robust, efficient, and user-friendly research tool.

- **Computational Efficiency**: A primary focus of the implementation was to optimise hardware utilisation, thereby reducing computational resource requirements and enhancing training and evaluation speeds. To achieve this, we prioritised a GPU-accelerated implementation leveraging the PyTorch framework, known for its efficient parallel processing capabilities in deep learning applications.
- **Code Readability**: To facilitate rapid development and integration within the machine learning and neuroscience communities, we placed significant emphasis on code readability. This was achieved through the use of descriptive variable naming conventions and comprehensive code documentation. Such practices ensured that researchers can easily understand, modify, and extend the codebase to suit their specific research needs.
- **Modular Architecture**: The implementation adopted a modular architecture, drawing inspiration from industry-standard practices. This approach involved dividing the system into distinct modules, each responsible for specific functionalities. The modular structure closely mirrors that of the PyTorch framework, providing an intuitive organisational system familiar to researchers experienced with PyTorch.
- **Parametric Flexibility**: A key feature of our implementation was its high degree of parametric flexibility. The system allowed for control over parameters at various levels of the model architecture. This design choice enables researchers to fine-tune the model with precision, accommodating a wide range of experimental setups.
- **Extensibility**: Recognising the rapid pace of advancements in the field, we designed the codebase with extensibility in mind. The implementation included clear documentation and adheres to clean coding practices, facilitating the seamless integration of future improvements and additional features.

3.1 Dataset Selection and Pre-Processing

3.1.1 Dataset Selection

CIFAR-10 was chosen as the competitive benchmark for Hebbian learning in our neural network implementations to analyse the differences that emerge in representations. MNIST and CIFAR-10 are two datasets which are used as a common benchmark in Hebbian research. While noticeable differences in the learnt representations and performance were encountered with the CIFAR-10 dataset, similar performances were achieved across previous methods when evaluated on MNIST, without any significant differences in the learnt representations.

The dataset is formed by 60,000 RGB images of size 32x32 pixels. There are 10 classes, which consist of animals and vehicles: airplane, automobile, bird, cat, deer, dog, frog, horse, ship, truck. There are 600 images per class, and the training set is formed by 50,000 images, while the test set is formed by the remaining 10,000 images.

3.1.2 Data Loading and Processing

Data processing consisted of implementing data augmentations to the dataset. Given its highlighted importance in research, ZCA whitening was implemented as an additional data

augmentation, altering the whole dataset when the data is initialised and loaded. ZCA in our implementation was calculated in the following manner:

Given a dataset X with n samples and d features, where $X \in \mathbb{R}^{n \times d}$, we computed the mean of the data:

$$\mu = \frac{1}{n} \sum_{i=1}^{n} X_i \tag{33}$$

The mean from each data point was subtracted to obtain the centred data matrix $\tilde{X}$:

$$\tilde{X} = X - \mu \tag{34}$$

The next step was to compute the covariance matrix Σ of the centred data:

$$\Sigma = \frac{1}{n-1} \tilde{X}^T \tilde{X} \tag{35}$$

We then performed Singular Value Decomposition (SVD) on the covariance matrix Σ:

$$U, S, V^T = \text{SVD}(\Sigma) \tag{36}$$

U is a matrix of eigenvectors (or singular vectors). S is a diagonal matrix containing the eigenvalues (or singular values).

To whiten the data, we needed to compute the whitening matrix. The whitening matrix W_{ZCA} was given by:

$$W_{\text{ZCA}} = U \cdot (S + \epsilon I)^{-\frac{1}{2}} \cdot U^T \tag{37}$$

where $(S + \epsilon I)^{-\frac{1}{2}}$ is the inverse square root of the eigenvalue matrix with a small regularisation term ϵ to ensure numerical stability. ϵ was a small constant added to the eigenvalues to prevent division by zero and affected the degree of whitening applied to the dataset.

Finally, we applied the ZCA whitening transformation to the centred data:

$$X_{\text{ZCA}} = \tilde{X} \cdot W_{\text{ZCA}} \tag{38}$$

The ZCA matrix for a dataset was stored to avoid calculation of the dataset statistics as it was a costly computational process due to the iteration across the whole dataset and the large matrix multiplication involved. This implementation leveraged PyTorch's capability to perform these matrix operations efficiently on GPU hardware, creating an efficient implementation which can also be used for other datasets.

The pre-computed ZCA matrix was applied to the data samples during data loading as a data augmentation. If ZCA-whitening was not used, the data was instead normalised using the dataset's statistics, which were also pre-computed for efficiency. A random horizontal flip was included for improved generalisability, and the image was converted into a tensor data structure.

For the configurations involving Hard-WTA competition, ZCA whitening was crucial for achieving high performance. Research on Hebbian rules and Hard-WTA competition noted a significant performance drop when ZCA whitening was not implemented. To provide equal variance

while preserving the spatial structure of the data, ZCA (Zero-phase Component Analysis) is a preprocessing technique which decorrelates features from the input and can improve convergence and learning in ANNs. The implementation is described in Appendix A.2.1

3.2 Neural Network Model Overview

The selected ANN architecture for this project was the Convolutional Neural Network. This choice was motivated by several factors including its structural similarity to the visual cortex, its proven efficacy in object recognition and detection tasks, and the existing body of research applying Hebbian learning principles to CNNs.

In addition to standard CNN architectures, this study incorporated variations inspired by both biological neural networks and advanced machine learning techniques. Specifically, we evaluated the application of Hebbian learning to architectures incorporating residual blocks, as popularised by ResNet and MobileNet, and Depthwise Separable Convolutions.

3.2.1 Code Structure and Organisation

The codebase was organised into several key components to facilitate efficient training, evaluation, and adherence to the project's design goals.

Experiment Management: Files responsible for handling the training and evaluation processes were prefixed with `experiment_X.py`, where X is the configuration name. These modules handled the instantiation and modification of models and datasets based on experiment-specific parameters. They also managed the visualisation method calls to understand the learning processes across different network layers. Accuracy evaluation was conducted at each epoch during the training of the backpropagation classifier layer.

Model Architecture Definition: Files prefixed with `model_X.py` defined the overall network architecture. These modules specified individual layers and their sequential interactions based on experiment parameters. They also included methods for filter visualisation. Various architectural configurations were defined here, including implementations that mirror state-of-the-art research models, as well as those incorporating residual blocks or depthwise separable convolutions.

Layer Implementation: To achieve modularity, individual CNN layers were implemented in files prefixed with `hebb_X.py`. This design mirrors the abstraction level found in PyTorch, encapsulating the computational processes within each layer. These modules contained all calculations related to forward passes and weight updates, including different learning rules and competition mechanisms. Separate implementations were provided for standard and depthwise convolutions.

Data Management: The `data.py` module was responsible for creating training and test datasets. It also managed the application of data augmentation techniques and implemented ZCA whitening calculations.

Visualisation Utilities: The `visualiser.py` module provided a suite of functions for visualising statistical metrics, visual representations, and neural activity within the CNNs. These visualisation tools can be called at various stages of the learning process (before, during, or after training) to provide a comprehensive view of learned features and neuronal changes. The `receptive_field.py` module handled the representation of the receptive field of neurons in different layers.

3.3 Hebbian CNN Implementation

3.3.1 Hebbian File Structure

The Hebbian implementation files, prefixed with `hebb_X.py`, shared a common framework that encapsulates the core functionalities of Hebbian learning within the neural network while following our design principles. The primary components of this framework were as follows:

- The `forward()` method served as the main point for processing input through a Hebbian layer. It called the appropriate method to calculate postsynaptic activity y, called the update function to compute weight changes based on the current input and activation and returned the postsynaptic activity to be used as input for upcoming layers in the network.
- The `compute_activation()` method was responsible for calculating the postsynaptic activity y through a convolutional operation. It performed a convolution operation to generate postsynaptic activity, applying any necessary modifications to the weights based on the experimental configuration. It returned both the postsynaptic activity and the potentially modified weights.
- The `compute_update()` method was the core of the Hebbian learning process, calculating weight changes based on the chosen learning rule and competition mode. Its structure was designed for flexibility and modularity, where each combination of learning rule and competition mode was treated as a unique mode within this method. Individual functional components of these rules and competition mechanisms were implemented as separate methods. The method computed and stored the weight changes to be applied to the layer's synaptic connections.
- The `local_update()` method applied the stored weight changes to the synapses, updating the synaptic connections at a local level.

The learning rules and competition modes described in the following sections were implemented in the `compute_update()` method.

3.3.2 Baseline: Hebbian Learning rule

As a baseline, a simple Hebbian learning rule without any competition mechanisms was implemented. While the original Hebbian rule defined in Equation 18 was the simplest to implement directly, Grossberg Instar's variation defined in Equation 19 was selected instead as it lets weights grow and decrease in strength and serves as the standard rule in Hebbian-CNN research.

In our implementation, the post-synaptic activities y were calculated through a convolutional operation between the weights w and pre-synaptic activities or layer-inputs x. An additional activation function can be applied to this result, such as a ReLu or Cosine Similarity function, as performed by Lagani, Falchi, Gennaro & Amato (2022). With y calculated, the weight changes were computed using weights w, presynaptic input x and postsynaptic activity y in a method called `update_basic_hebbian()`. The change in weights Δ_w was stored in a temporary variable buffer for computational efficiency and posterior data analysis.

Weights were updated in our model using the stored weight changes. The `local_update()` method incorporated the weight changes as a gradient, which can then be combined with backpropagation-calculated gradients for a hybrid approach with an optimiser or used in a fully Hebbian approach. This was controlled by parameter α, where $\alpha = 1$, used in all our Hebbian experiments, dictated a fully Hebbian approach.

Our implementation for the calculated Δ_w for this rule is detailed in the following section, as it serves as the fundamental learning rule in all configurations except BCM learning mode. The

next step in the project was the state-of-the-art (SOTA) SoftHebb and Hard-WTA Hebbian learning rules as they presented the best performance through approaches involving different competition dynamics between neurons.

3.3.3 Hard-WTA Research Issues

The code implemented by Amato et al. (2019) was selected as our main inspiration, and we started with Hard-WTA competition. While both Amato et al. (2019) and Miconi (2021) used local Hebbian learning, Miconi (2021) always required the usage of a backpropagation optimiser as it used gradients, while Amato et al. (2019) allowed for training without an optimiser or a combination with backpropagation gradients.

The implementation by Amato et al. (2019) explicitly unfolded the image into patches. This operation is resource intensive, requiring increased memory requirements due to unfolding of tensor variables, reshaping and permuting dimensions of variables, and creation of variables based on the memory-intensive unfolded tensors. It also used non-matrix multiplication which is not as optimised for GPU hardware as matrix-based operations. It applied competition to neurons in the same spatial location but different channels, encouraging each filter to learn a different representation. These issues can be seen in the code repository for Amato et al. (2019).

3.3.4 Optimised Hard-WTA

We used the Conv2D operation provided through PyTorch as it allowed for optimised matrix operations to take place with implicit patches, working directly with the input without needing to unfold it into patches. Additional creation of variables was limited to avoid intense memory allocations. This represented our main step to optimise Hebbian learning with GPU operations, inspired by the implementation by Journé et al. (2022) and theoretical advancements behind FastHebb (Lagani, Gennaro, Fassold & Amato 2022), which used implicit patches and matrix operations.

These optimisations allowed for scalability with input size, improved memory allocation, reduced computations performed and faster speeds due to processing smaller variables. Memory was now constrained closer to the original input size, rather than significantly increasing the input memory requirements. The use of matrix-based operations allowed for efficient usage of GPU devices, increasing training speeds.

The Hard-WTA competition for selecting winner neurons in the `compute_wta_mask()` method first flattened the spatial dimensions into a single dimension and uses `argmax()` PyTorch function to locate the maximum value activation along the channel dimension for each spatial location. A binary mask was created, where winners are only preserved, and applied to the original activities.

With this masked activation acting as the postsynaptic activity y, the Grossberg Instar rule in Equation 19 was used in the `update_hardwt()` method. We applied A convolution between presynaptic input x and weights w for the yx term. For the yw term, a single value for each output channel representing the total activation of that channel across all examples in the batch and all spatial locations was calculated and applied to the weights. The result was normalised to avoid large changes in the weights.

We note yw term lost some locality in its updates as it sums over the spatial dimensions. It still served a similar purpose to the original rule, applying a modulation of the weight decay term, but applied a more global normalisation across batch and spatial dimensions at each channels in a layer. This change was implemented for computational efficiency and functionality in this convolutional context, and was also applied by Journé et al. (2022).

Cosine similarity between presynaptic activity x and synaptic connections w was also implemented

as an additional activation function during forward passes and during weight updates, as performed by Amato et al. (2019).

3.3.5 Optimised SoftHebb

We implemented SoftHebb by applying the soft competition to the postsynaptic activity using a Softmax function and an inverse temperature parameter to control the competition. We did not require additional optimisation as Journé et al. (2022) already used our optimisation techniques. The implementation can be seen in Appendix A.2.2.

The following step was to apply antihebbian learning to all neurons except the maximally activated unit. All neurons were initialised as antihebbian, and the index of the highest activation was used to convert it back to a Hebbian update. These competition mechanisms were implemented in the `compute_softwta_activations` method.

The same Grossberg Instar implementation detailed previously was applied to calculate weight changes. The only difference was the use of the result of the competition in the yx term instead of the masked activations, as detailed in Equation 32 and in the `update_softwta()` method.

3.3.6 BCM Learning Rule

Our implementation for the BCM learning rule from Equation 23 embraced the matrix-based and resource optimisation approach. An optional competition mode was applied to the postsynaptic activities for increased competition. Hard-WTA competition selection used in Section 3.3.4 was used as a competitive baseline.

The threshold θ was updated using an exponential moving average of the WTA activations. The parameter *theta_decay* determined the change in θ based on recent activity. The non-linearity $\psi(y - \theta)$ was computed as the product of the postsynaptic activity and the difference between this activity and the updated θ threshold. Each channel had its own θ threshold.

The final step in our implementation consisted of using a convolutional operation to calculate the correlation of presynaptic input x with non-linearity $\psi(y - \theta)$. The updates were normalised to maintain consistency with the other learning rules. The BCM implementation was computed in the `update_bcm()` method and can be seen in Appendix A.2.2.

3.3.7 Temporal and Statistical Thresholds

We created two distinct temporal and statistic competition modes inspired by the biological mechanisms of metaplasticity and homeostasis to expand on the concept of competition mechanisms.

Temporal Selection: Temporal competition was inspired by the concept of synaptic traces and metaplasticity, where the threshold for plasticity changes based on a history of activity called synaptic traces which stored in neuron. This phenomenon is known as metaplasticity. It promotes stability in representations learnt over time, selecting which neurons to activate when features appear consistently across time steps.

We updated synaptic traces based on recent activations in the `update_activation_history()` method, storing the post-synaptic activities as a moving average where a window parameter controlled the amount of activities stored. With the updated synaptic trace, we calculated the winners which surpass an activity threshold based on the synaptic trace. The threshold, calculated in the `compute_temporal_winners()` method, was a mean of the median activations in the synaptic trace favouring neurons which had been active consistently across time.

We then applied this competition mask back to the postsynaptic activities. An additional competition mode, such as Hard-WTA described in Section 3.3.4, could be applied for further specialisation. Finally, the Grossberg Instar implementation was used to calculate synaptic changes in the `update_temporal_competition()` method.

Statistical Selection: Adaptive threshold competition was inspired by homeostatic plasticity, which adjusts neural plasticity to maintain stable levels of activity. This avoided dominance of neurons and the vanishing activities of neurons. Typically homeostasis is implemented directly based on the firing rate and activations of neurons, as seen in Appendix A.2.2.

Our direct implementation of homeostasis maintained a running average of each neuron's activity and compares this to a target activity level, where parameters control the influence of recent activity and activities stored in the synaptic trace. Synaptic weights were gradually scaled up or down to bring each neuron's average activity closer to the target and to modify their sensitivity to inputs, as seen in the `synaptic_scaling()` method.

An indirect approach was to use the input statistics to govern the competition while maintaining stable activity across neurons. First similarities between weights and input were calculated, using a simple form of cosine similarity.

The threshold for competition calculated in the `compute_adaptive_threshold()` method was based on the similarity scores. If input patterns produced high similarity scores, the threshold increased, while low similarity scores decreased the threshold. This mechanism indirectly produced homeostasis by preventing over-activation when many high similarity scores occurred through a high threshold which restricts which neurons were updated. It prevented under-activation when low similarity scores occurred by lowering the threshold to facilitate the update of neurons.

This mechanism allowed for faster changes to unseen distributions, allowing more neurons to adapt and mimic sensory adaptation in biological systems. The threshold was computed based on the mean and standard deviation of the current input batch, allowing for rapid adaptation to changes. If the overall strength of inputs increased, the mean similarity score would increase, raising the threshold and promoting specialisation. If the variability of inputs changed, the standard deviation similarity score reflected these changes and affected the threshold to allow more neurons to adapt.

Winner neurons were selected with this threshold and optionally passed to a further competition mechanism. This additional competition mechanisms in both modes promoted efficient specialisation and sparse connectivity. The Grossberg Instar rule was again used to update the synaptic connections in the `update_adaptive_threshold()` method.

3.3.8 Pre-synaptic Competition

We designed a form of pre-synaptic competition, following the suggestions by Lagani (2024). The pre-synaptic input x was combined with parallel synaptic coupling $m_{i=1} = 1/w1$, $m_{i=2},...,m_{i=N}$ where i is the coupling to each neuron and N the total number of neurons. The concept of synaptic coupling is similar to weights in neural networks whereas in biological neural networks they refer to the strength of connections between neurons. However, unlike synapses they directly influenced each other, leading to competitive dynamics in neural signalling and learning.

In the context of CNN, synaptic couplings were represented as the kernel weights connecting input channels to output channels. Based on the recommendations from Lagani (2024), the weights were converted into synaptic couplings m by first inverting their value, to provide normalisation, competition and simulate electrical admittances:

$$m = \frac{1}{w} \tag{39}$$

A competition mechanism was applied to these couplings in the `compute_activations()` method and the result was used for convolutions and update calculations. The different competition modes which can be selected to introduce presynaptic competition were implemented in the `compute_presynaptic_competition()` method and are listed as:

1. **Linear Competition, defined as:**

$$w_{eff} = \frac{m}{\sum_i m_i + \epsilon} \tag{40}$$

   where $m = \frac{1}{|w|+\epsilon}$, w is the original weight, and ϵ is a small constant (1e-6) to prevent division by zero. It normalises the weights to values between 0-1.

2. **Softmax Competition, defined as:**

$$w_{eff} = \text{softmax}(m) = \frac{e^{m_i}}{\sum_j e^{m_j}} \tag{41}$$

   where $m = \frac{1}{|w|+\epsilon}$ as before. This creates a more pronounced competition where stronger connections are emphasised, and the weights summed together add up to 1.

3. **L2 Norm Competition, defined as:**

$$w_{eff} = \frac{m}{\sqrt{\sum_i m_i^2}} \tag{42}$$

   where $m = \frac{1}{|w|+\epsilon}$. This ensures that the sum of squared effective weights equals 1.

In all cases, the competition was applied along the output channel dimension. This means that for each input channel and same spatial location in the kernel, the weights connecting to all output channels competed with each other. This promotes diversity among output channels, encouraging them to specialise in detecting different features from the same input locations.

Additional competition across different spatial and channel dimensions of the architecture were also implemented. Applying competition across spatial dimensions within each input-output channel pair encouraged each input-output channel pair to focus on specific spatial patterns. This was implemented in `compute_presynaptic_competition_spatial()` method.

Implementing competition across input channels for each output channel and spatial location encouraged each output neuron to be selective about which input features it responds to. This was implemented in `compute_presynaptic_competition_input()` method.

A more global competition across all dimensions simultaneously created a more intense competition where every weight competed with all others leading to very sparse but highly specialised connections. This was implemented in `compute_presynaptic_competition_global()` method.

3.3.9 Lateral Inhibition

We created a fixed kernel created through a difference of Gaussians (DoG) function to simulate the effect of lateral inhibition and surround modulation found in early layers of the visual cortex and the excitatory/inhibitory impact on neighbouring neurons. It was implemented in the `create_sm_kernel()` method.

This kernel models how neurons in the visual cortex respond to stimuli in their receptive field and surrounding areas. The centre (excitatory) region represents the classical receptive field,

while the surround (inhibitory) region models lateral inhibition from neighbouring neurons as seen in Figure 7. It is mathematically defined as:

$$K_{SM}(x,y) = \frac{1}{K_{center}} \left(\frac{G_e(x,y)}{2\pi\sigma_e^2} - \frac{G_i(x,y)}{2\pi\sigma_i^2} \right)$$

where:

$$G_e(x,y) = \exp\left(-\frac{x^2+y^2}{2\sigma_e^2} \right)$$

$$G_i(x,y) = \exp\left(-\frac{x^2+y^2}{2\sigma_i^2} \right)$$

$$K_{center} = \left(\frac{G_e(x,y)}{2\pi\sigma_e^2} - \frac{G_i(x,y)}{2\pi\sigma_i^2} \right) \Bigg|_{x=0,y=0}$$

where $K_{SM}(x,y)$ represents the surround modulation kernel at position (x,y). $G_e(x,y)$ and $G_i(x,y)$ are the excitatory and inhibitory Gaussian functions, respectively. σ_e and σ_i are the standard deviations for the excitatory and inhibitory Gaussians. K_{center} is the value of the unnormalised kernel at the centre (0,0). The kernel was computed as the difference between the normalised excitatory and inhibitory Gaussians. This kernel strengthened synapses in the immediate neighbourhood of a neuron, and weakened synapses further from this neighbourhood.

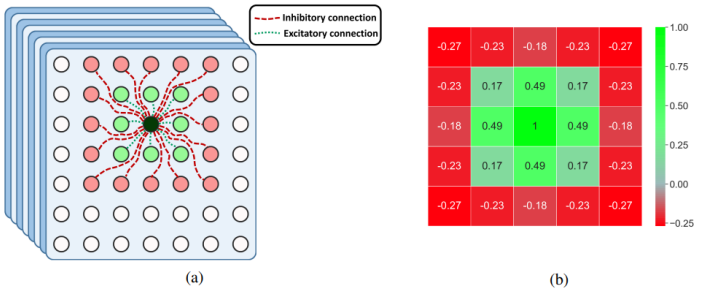

Figure 7: Surround modulation kernel, as presented in Hasani et al. (2019)

This fixed kernel was applied to the postsynaptic activities through a convolutional operation, enhancing contrast and edge detection while suppressing uniform backgrounds.

3.3.10 Depthwise Separable Convolution

We developed a novel Hebbian implementation of the biologically plausible Depthwise Separable Convolution to investigate the potential of biological rules which can generate centre-surround kernels observed in neurons within the visual cortex.

The depthwise convolution shared similarities with the previously described convolutional approach, with modifications in the application of updates and competition mechanisms to ensure filter independence. Given the one-to-one correspondence between input and output channels, we extracted this relationship from the convolutional operation output yx.

By extracting the diagonal elements along the input and output channel dimensions, we selected only the elements where the input channel index matches the output channel index, preserving the independence criteria. The yw term was also adjusted to preserve independence. In the

depthwise CNN, yw was computed separately for each channel, ensuring the update for each filter depended only on its own activations.

The competition modes were modified to shift the dimension of competition from neurons at the same spatial location across channels to neurons within the same channel across spatial locations. Additionally, to maintain independence, the number of output channels was required to match the number of input channels. These implementation modifications can be seen in Appendix A.2.3.

The pointwise convolution, implemented as a 1×1 convolution, was fully compatible with the previously established learning rules and competition mechanisms. This compatibility stems from the interaction between channels at identical spatial locations and lack of filter independence. The implementation required only a kernel size of 1.

This variation of the Hebbian-CNN was located in `hebb_depthwise.py`, and the model using this architecture was defined in `model_depthwise.py`, as shown in Appendix A.2.

3.3.11 Residual Block

We implemented additional feedforward connections linking layers at different sequential orders to enhance the biological plausibility of the model. This approach was inspired by the concept of skip connections and residual blocks found in advanced deep CNN architectures such as ResNet and MobileNet.

Convolutional blocks were constructed with a bottleneck structure utilising Depthwise Separable Convolutions for feature extraction. A skip connection was incorporated to link the input of the block to the output of the bottleneck. This block design is related to the approach employed in MobileNet, using depthwise separable convolutions for parameter reduction while integrating the advantages of residual blocks from ResNet to facilitate training of deeper networks.

The block employed an inverted bottleneck structure, built by three convolutional layers. The first layer was a pointwise convolution that expanded the number of channels, facilitating the learning of rich channel features. The second layer was a depthwise convolution which learnt the spatial information from the expanded feature space. The third was another pointwise convolution that projected the features back to a desired output channel count.

The skip connection was a pointwise convolution designed to match the input size to the output size of the third layer of the bottleneck. The output of this skip connection was added to the output of the bottleneck structure, enabling more efficient learning in deeper architectures. The block module was defined in the `HebbianResidualBlock()` class in (model_residual.py), as seen in Appendix A.2.

3.3.12 Dale's Principle Network

We modified the weight initialisation and update processes to investigate the learning dynamics of a purely excitatory-driven network that follows Dale's Principle, in comparison to mixed biologically implausible excitatory-inhibitory weights.

All synaptic weights were initialised to positive values with an absolute value function to ensure compliance with this principle. In any operation involving weights as a term, the absolute value function was applied to maintain this constraint. The change in weights can still produce both LTP and LTD, as both directions are essential for plasticity.

If a weight update resulted in a potentially inhibitory weight, two approaches were considered, the application of an absolute value function to ensure exclusive excitatory impact or a ReLU function to allow only excitatory connections to exist. This approach may lead to neurons

with activity levels close or equal to zero remaining near this level of activity during learning, potentially creating sparse connections at the expense of facilitating the formation of distributed representations.

To address this phenomenon, two alternative mechanisms were proposed, a homeostasis mechanism as detailed in Section 3.3.7 to maintain all neurons active or a probabilistic metaplasticity mechanism that pruned synapses near a local minimum and generated new synapses based on local activity. These alternatives could potentially mitigate the sparsity issue while maintaining the biological plausibility of the network.

This Hebbian variation and its mechanisms which respects Dale's Principle were created in the `hebb_abs.py` and `hebb_abs_depthwise.py` files, further showcased in Appendix A.2.5.

3.3.13 Visualisation Learning

We developed a comprehensive suite of visualisation tools to enhance our understanding of Hebbian learning and neuronal feature detection in CNNs. These can be employed at any time during learning or for posterior analysis. These tools, detailed in Appendix A.2, provided innovative insights into the learning dynamics of Hebbian-CNN across various scales, from individual synapses to entire layers. All plots are stored using WanDB.

Descriptive statistical information was compiled for all neurons within each layer, encompassing both the pre-update weights and the weight changes. This analysis included measures such as mean, median, standard deviation, and the ratio of positive to negative weights or weight updates. The data was presented for a selected subset of neurons as well as for the entire layer, offering a multi-scale perspective on neuronal behaviour.

Violin plots were employed to depict the distribution of weight changes for a selected subset of neurons, providing a nuanced view of synaptic plasticity. Additionally, weight value distributions across entire layers were visualised to understand neuronal connectivity patterns and synaptic sparsity in different model configurations. These distributions also offered insights into the stability of weight configurations over time.

An innovative approach to visualising CNN filters was developed, focusing on weight changes per spatial location within each filter. This technique involved plotting the mean weight change across channels, resulting in a 2D representation that reveals emergent patterns and localised regions of LTP and LTD. For 1x1 convolutions, which lack spatial structure, bar plots were utilised to represent weight changes, with colour coding to indicate magnitude and direction of change.

Complementing this, a global LTP/LTD visualisation was created to illustrate the dominant learning direction for a selected number of neurons, providing a global view of synaptic plasticity across the network. Additionally, the features from the final convolutional layer were reduced to a 2D space through dimensionality reduction techniques such as UMAP to allow a visual representation of the different classes and their clusters. This enabled us to determine whether the model effectively groups inputs of the same class together while distinguishing them from inputs of different classes. These described tools were defined in the `visualiser.py` file.

The filters themselves were also directly visualised to provide insight into the features they represent. For the initial convolutional layer, full RGB visualisations offered an intuitive representation of low-level features. In deeper layers, where direct visualisation is less interpretable, the mean across channels was used to create 2D representations that only capture dominant patterns. It was defined in the `visualize_filters()` method in all `model_X.py` files.

To further comprehend neuronal selectivity, receptive fields were visualised using a Projected Gradient Ascent (PGA) method. In biological terms, the receptive field corresponds to the

region of sensory space to which a neuron responds. Receptive fields in CNNs are hierarchical and grow larger in deeper layers due to the cumulative effect of convolutional operations. This hierarchical structure should allow the network to capture increasingly complex and abstract features as information propagates through the layers.

In the first layer, receptive fields may correspond to simple edge detectors or spots of colour which match with the direct filter visualisations at the first layer, while in deeper layers they may represent complex object shapes or even entire objects which were not captured by the direct filter visualisations of deeper layers which only focus on patterns.

The PGA technique optimised an input image initialised with random noise and sized to match the neuron's receptive field, to maximise the neuron's activation. We forward propagated this image through the network to the neuron of interest and computed the gradient of the neuron's activation with respect to the input image. Next, we updated the input image to maximise the neuron's activation, subject to regularisation constraints to keep the image interpretable.

This process revealed the optimal stimuli for neurons at various levels of the architecture, demonstrating the hierarchical representations formed across layers. We implemented a custom implementation of PGA in the `receptive_fields.py` file, which incorporated L2 regularisation and gradient optimisation to aid with finding the optimal and explainable stimuli which activates a neuron.

3.4 Experimental Setup

3.4.1 Configurations

Tables 1 and 2 present a detailed overview of various configurations designed to evaluate competition dynamics and Hebbian learning rules in neural networks. We designed and ran each configuration to systematically explore different combinations of architectural designs, competition mechanisms, and learning rules, each designed to investigate specific aspects of neural network behaviour and performance.

The configurations primarily utilised two main architectural approaches: the SoftHebb architecture, based on the work of Journé et al. (2022), which incorporated padding and has larger number of filters per layer, and the Lagani, Falchi, Gennaro & Amato (2022) padding-free and smaller architecture with 3 or 4 convolutional layers. These architectural choices provided a foundation for comparing different network structures and their impact on learning and competition dynamics. More details for these architectures can be found in Appendix A.3.

A key focus of this experimental setup was the examination of various competition mechanisms. The configurations incorporated a range of approaches, including Soft Winner-Take-All (SoftWTA), Hard Winner-Take-All (HardWTA), presynaptic competition, temporal selection, statistical selection, and surround lateral inhibition. These mechanisms were combined, as seen in Table 1 where configurations like `SoftHebb-Presynaptic/HardWTA/Cos-Instar` integrated both HardWTA and presynaptic competition while using a Cosine Similarity Activation. This approach allowed for an understanding of how different competition strategies interact and influence network behaviour.

The learning rules employed in these configurations are predominantly the Grossberg Instar rule and the Bienenstock-Cooper-Munro (BCM) rule. The inclusion of a configuration using backpropagation with `SoftHebb-Backpropagation` provided a benchmark for comparison with traditional gradient-based learning methods. This diversity in learning rules enabled a comparative analysis of their effectiveness in similar scenarios.

Table 1: Hebbian Learning Configurations and Features

Configuration	Architecture	Competition Mechanism	Learning Rule	Additional Features
`SoftHebb-SoftWTA-Instar`	SoftHebb	Soft WTA	GI	Hebb. & anti-Hebb. learning
`Lagani-HardWTA/Cos-Instar`	Lagani	Hard WTA	GI	Cos. sim. activation
`Lagani_Deep-HardWTA/Cos-Instar`	Lagani_Deep	Hard WTA	GI	Cos. sim. activation
`SoftHebb-HardWTA/Cos-Instar`	SoftHebb	Hard WTA	GI	Cos. sim. activation
`SoftHebb-HardWTA/Cos-BCM`	SoftHebb	Hard WTA	BCM	Cos. sim. activation
`SoftHebb-None-Instar`	SoftHebb	None	GI	-
`SoftHebb-Pre/HardWTA/Cos-Instar`	SoftHebb	Hard WTA+Pre	GI	Cos. sim. activation
`SoftHebb-Temp/HardWTA/Cos-Instar`	SoftHebb	Hard WTA+Temp	GI	Cos. sim. activation
`SoftHebb-Stats/HardWTA/Cos-Instar`	SoftHebb	Hard WTA + Stats	GI	Cos. sim. activation
`SoftHebb-Sur/HardWTA/Cos-Instar`	SoftHebb	Hard WTA+Sur	GI	Cos. sim., Sur lat. inhib.
`Depthwise_SoftHebb-Sur/HardWTA/Cos-Instar`	SoftHebb	Hard WTA+Sur	GI	Cos. sim., Depthwise sep. conv.
`Residual_SoftHebb-Sur/HardWTA/Cos-Instar`	SoftHebb	Hard WTA+Sur	GI	Cos. sim., Res. connections
`Dale_SoftHebb-Sur/HardWTA/Cos-Instar`	SoftHebb	Hard WTA+Sur	GI	Cos. sim., Dale's Principle
`SoftHebb-Backpropagation`	SoftHebb	N/A	BP	-

Note: WTA: Winner-Take-All, GI: Grossberg Instar, BCM: Bienenstock-Cooper-Munro, BP: Backpropagation, Pre: Presynaptic, Temp: Temporal, Stats: Statistical, Sur: Surround, Cos. sim.: Cosine similarity, sep. conv.: separable convolutions, Res.: Residual, lat. inhib.: lateral inhibition, Hebb.: Hebbian

Table 2: Purposes of Hebbian Learning Configurations

Configuration	Purpose
`SoftHebb-SoftWTA-Instar`	Evaluate SoftHebb research
`Lagani-HardWTA/Cos-Instar`	Evaluate Lagani Hard-WTA research
`Lagani_Deep-HardWTA/Cos-Instar`	Evaluate Lagani Hard-WTA 4-layer research
`SoftHebb-HardWTA/Cos-Instar`	Evaluate Hard-WTA in network with padding
`SoftHebb-HardWTA/Cos-BCM`	Compare BCM to Grossberg-Instar rule
`SoftHebb-None-Instar`	Evaluate basic learning without competition
`SoftHebb-Pre/HardWTA/Cos-Instar`	Assess impact of presynaptic learning
`SoftHebb-Temp/HardWTA/Cos-Instar`	Evaluate temporal competition with/without WTA
`SoftHebb-Stats/HardWTA/Cos-Instar`	Evaluate statistical competition with/without WTA
`SoftHebb-Sur/HardWTA/Cos-Instar`	Assess effect of surround modulation kernel
`Depthwise_SoftHebb-Sur/HardWTA/Cos-Instar`	Evaluate depthwise equivalent of hard-WTA research
`Residual_SoftHebb-Sur/HardWTA/Cos-Instar`	Evaluate residual equivalent of hard-WTA research
`Dale_SoftHebb-Sur/HardWTA/Cos-Instar`	Evaluate Dale's Principle in hard-WTA research
`SoftHebb-Backpropagation`	Evaluate backpropagation in SoftHebb models

Configurations following `SoftHebb-Sur/HardWTA/Cos-Instar` and onward incorporated additional features to enhance network performance or investigate specific neural phenomena. The cosine similarity activation, present in most configurations, was used as advantages were noted by Lagani, Falchi, Gennaro & Amato (2022). Some configurations explored architectural modifications, such as depthwise separable convolutions (`Depthwise_SoftHebb-Sur/HardWTA/Cos-Instar`) and residual connections (`Residual_SoftHebb-Surr/HardWTA/Cos-Instar`), implemented after the first layer while maintaining input and output channel equivalence to standard CNN layers. The `Dale_SoftHebb-Surr/HardWTA/Cos-Instar` configuration introduced biological constraints by adhering to Dale's Principle, offering insights into the impact of neuron-type restrictions on network performance.

This systematic experimental setup provided a foundation for understanding how various components interact and contribute to network performance, potentially leading to the development of more efficient and biologically plausible neural network models.

3.4.2 Training and Testing Details

The training stage for Hebbian configurations followed a sequential approach comprising three distinct phases: feature learning, feature extraction and analysis, and classifier training. In the initial feature learning phase, only the convolutional layers were trained for a single epoch to develop feature representations. For all configurations, a batch size of 64 datapoints was used to train the models.

For Hard-WTA configurations, a learning rate of 0.1 was applied to it, while for Soft-WTA the custom learning rate schedule detailed in Journé et al. (2022) and in Appendix A.2 was used instead. It is noted that configurations employing `SoftWTA` and those using `HardWTA` or no competition mechanism (`None`) utilised slightly different parameters. These variations included the application of ZCA-Whitening. The specific details of these training configurations are described in Appendix A.2 and A.3.

Throughout the Hebbian training process, we employed our custom-visualisation tools to store and visualise layer data. This method allowed for a comprehensive analysis of the network's developing representational abilities. Once the feature learning phase was completed, the convolutional layer weights were frozen to retain the learned features. At this stage, we gathered more information to obtain a deeper understanding of the representations learnt at different layers.

The final phase of our training procedure involved the classifier head, which was trained on top of the frozen convolutional layers. This classifier was trained through backpropagation for five epochs through an Adam optimiser with a learning rate of 0.001. To ensure rigorous evaluation, we assessed both the training and test sets at each epoch, storing the results for comparative analysis between different configurations.

The whole backpropagation model was trained through an Adam optimiser with a learning rate of 0.01 and no data-whitening. All other hyperparameters remained the same as in Hebbian configurations.

3.4.3 Evaluation Metrics

We employed the following evaluation metrics to facilitate a quantitative comparison between Hebbian learning configurations: accuracy, precision, recall, and F1 score.

In classification tasks, True Positives (TP) refer to instances where the model correctly identifies a positive class accurately. False Negatives (FN) occur when the model incorrectly predicts a positive class as negative. On the other hand, True Negatives (TN) are cases where the model

correctly identifies negative instances correctly. Lastly, False Positives (FP) represent cases where the model incorrectly predicts a negative instance as positive. These four terms are essential for calculating performance metrics .

Accuracy serves as a fundamental measure of the model's overall correctness across all classes. While accuracy provides a good indication of performance, it may yield misleading results for imbalanced datasets. Equation 43 expresses this mathematically:

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN} \tag{43}$$

Precision quantifies the model's ability to avoid labelling negative samples as positive. This metric is particularly crucial in applications where false positives incur significant costs or risks. Equation 44 describes this mathematically:

$$\text{Precision} = \frac{TP}{TP + FP} \tag{44}$$

Recall reflects the model's capacity to identify all positive instances within the dataset.This metric is important in scenarios where false negatives are particularly undesirable or costly. Equation 43 expresses this mathematically:

$$\text{Recall} = \frac{TP}{TP + FN} \tag{45}$$

The **F1 score** provides a balanced measure of the model's performance. By combining precision and recall into a single metric, the F1 score offers a holistic view of the model's performance. Equation 46 represents this mathematically:

$$\text{F1-Score} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \tag{46}$$

4 Results

4.1 Initialisation of Weights

The weights for each synaptic connection were initialised using the same weight distribution, as both Soft-WTA and Hard-WTA functioned well with this scheme. A normal random distribution with a large radius range was used, as low radius range did not produce learning in the SoftHebb model, detailed in Journé et al. (2022).

$$\text{weight\_range} = \frac{25}{\sqrt{C_{in} \cdot K_h \cdot K_w}}$$

$$W = \text{weight\_range} \cdot X$$

$$X \sim \mathcal{N}(0, 1)$$

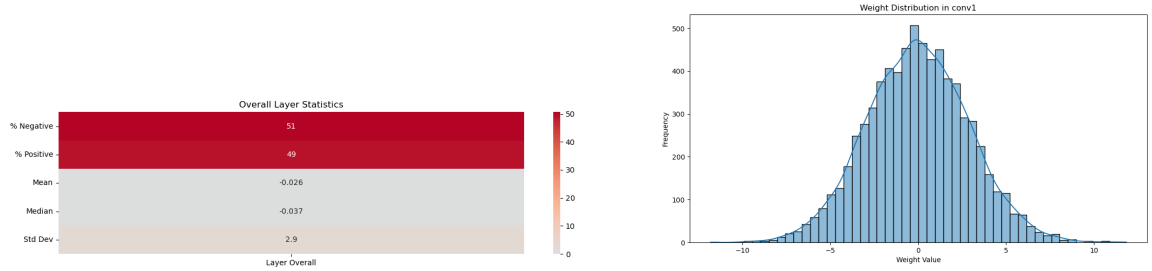

Figure 8: Results of the initial statistics and normal distribution of weights in the first layer. Deeper layers also present similar initial statistics and distribution.

The obtained standard normal distribution was centred at 0, so approximately half of the generated values were negative and half positive, before being scaled by weight_range to create a large range of weight values. Statistical results for the initial weights and the initial distribution for the first convolutional layer can be visualised in Figure 8, visualised using the tools we developed for the project.

4.2 Configurations Quantitative Results

Each experimental configuration detailed in Table 1 was evaluated with a fixed random seed for equal experimental conditions, and the mean Accuracy and F1-score metrics for the test set are presented in Table 3.

`SoftHebb-SoftWTA-Instar` and `SoftHebb-Surround/HardWTA/Cos-Instar` demonstrated superior performance, with 73.6% and 73% accuracy respectively. With these results, we achieved a new milestone and state-of-the-art performance for Hard-WTA competitino, significantly advancing the field of Hebbian learning in neural networks. State-of-the-art (SOTA) includes all results surpassing 70% accuracy.

In addition, the new depthwise architecture in `Depthwise_SoftHebb-Surround/HardWTA/Cos-Instar` achieved competitive results with notably less parameters, reducing parameters by 6.6x from 5,893,162 to 883,978 while achieving 69% accuracy. Other configurations including `SoftHebb-HardWTA/Cos-BCM`, `SoftHebb-Temp/HardWTA/Cos-Instar`, `Residual_SoftHebb-Surround/HardWTA/Cos-Instar` and `SoftHebb-HardWTA/Cos-Instar` surpassed 70% accuracy through a different learning rule, competition mechanism, architectural change or dependence on a sole competition mechanism respectively.

Table 3: Performance Metrics for Experimental Configurations on same Fixed Seed

Configuration	Accuracy (%)	F1 Score
SoftHebb-SoftWTA-Instar	**73.6**	**0.738**
Lagani-HardWTA/Cos-Instar	57.7	0.571
Lagani_Deep-HardWTA/Cos-Instar	53.2	0.527
SoftHebb-HardWTA/Cos-Instar	72.1	0.72
SoftHebb-HardWTA/Cos-BCM	71.1	0.711
SoftHebb-None-Instar	19.9	0.6
SoftHebb-Presynaptic/HardWTA/Cos-Instar	66.5	0.664
SoftHebb-Temp/HardWTA/Cos-Instar	71.5	0.715
SoftHebb-Stats/HardWTA/Cos-Instar	59.8	0.582
SoftHebb-Surround/HardWTA/Cos-Instar	**73**	**0.73**
Depthwise_SoftHebb-Surround/HardWTA/Cos-Instar	69	0.689
Residual_SoftHebb-Surround/HardWTA/Cos-Instar	71	0.708
Dale_Depthwise_HardHebb-Surround/HardWTA/Cos-Instar	62	0.615
SoftHebb-Backpropagation	70.8	0.704

The remaining configurations attained results closer to previous Hebbian-CNN literature, around 60% accuracy, with the low accuracy of 19% of `SoftHebb-None-Instar` demonstrating the necessity of competition mechanisms.

This high accuracy level in our best configurations was achieved by training for only 5 epochs rather than the 50-20 epochs employed in research, and further improvements can be obtained when allowing the classifier to converge its weights. Training a model for longer epochs can improve performance, but it can also conceal the expected accuracy for shorter training regimes. For comparison, Journé et al. (2022) achieved 80.3% accuracy after training the classifier for 50 epochs, Miconi (2021) achieved 64.6% accuracy after training the classifier for 20 epochs, and Lagani, Falchi, Gennaro & Amato (2022) achieved 64.43% under 20 epochs.

To ensure robustness and stability against randomness, the best models in terms of performance and parameters were evaluated in two additional runs with different random seeds. One of the runs was evaluated for 10 epochs to showcase improved performance could be achieved with longer training times.

Table 4: Accuracy for Experimental Configurations on different Fixed Seeds per Run

Configuration	Run 1 (%)	Run 2 (%)	Run 3 (%)	Mean (%)	10 Epoch Acc.(%)
SoftHebb-SoftWTA-Instar	73.6	75	72.6	**73.73**	**75.2**
SoftHebb-Surr/HardWTA/Cos-Instar	73	73.35	73.48	**73.27**	**74.29**
Depthwise_SoftHebb-Surr/HardWTA/Cos-Instar	69	68.9	69	68.97	70.5

Table 4 presents these robustness results. `SoftHebb-SoftWTA-Instar`, `SoftHebb-Surround/HardWTA/Cos-Instar` and `Depthwise_SoftHebb-Surround/HardWTA/Cos-Instar` maintained their high performance, and across these runs the largest difference was 1.27% accuracy from the mean for the `SoftHebb-SoftWTA-Instar` configuration. When allowing the classifier to train for 10 epochs, improvements of up to 2% in the accuracy were reached, indicating training the classifier for longer epochs can further improve the final results.

Figure 9 shows the results obtained from configuration `SoftHebb-SoftWTA-Instar` (blue line) and `SoftHebb-Surround/HardWTA/Cos-Instar` (red line). In both cases, similar performance was observed both during training (left panel) and testing (right panel), with a fast accuracy increase at each epoch. Learning was remarkably faster in our Hebbian configurations than

with backpropagation (yellow line) on both train and test data under the shorter training times, with the Hebbian configurations presenting steep slope increases at each epoch compared to the consistently gradual increases in backpropagation, a sign of fast learning.

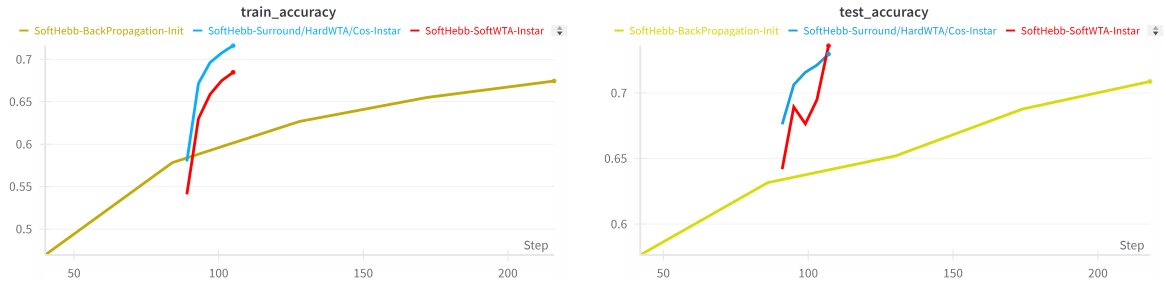

Figure 9: Train and Test accuracy of **SoftHebb-SoftWTA-Instar** and **SoftHebb-Surround/HardWTA/Cos-Instar** configurations compared to Backpropagation during 5 epochs. Epochs are marked incorrectly as WanDB handles steps/epochs inaccurately.

The distribution of weights varied depending on the competition mode and learning rule used, as plotted in Figures 10-12. SoftWTA (Figure 10) produced a distribution with a pronounced central peak and relatively narrow range, closely adhering to its initial Gaussian distribution, indicating only small but distributed changes were made to the original weights. Hard-WTA with the Grossberg Instar rule (Figure 11) resulted in a similar shape but with a wider spread of values. The BCM learning rule (Figure 12) led to a distinct distribution characterised by a sharp peak near zero and heavy tails, indicating a tendency towards weight sparsity while also allowing for some large positive and negative weight values. Deeper layers for these configurations aligned with their respective distribution tendencies.

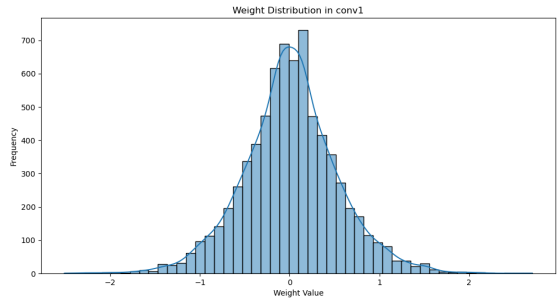

Figure 10: Weight distribution at first layer characteristic of Soft-WTA.

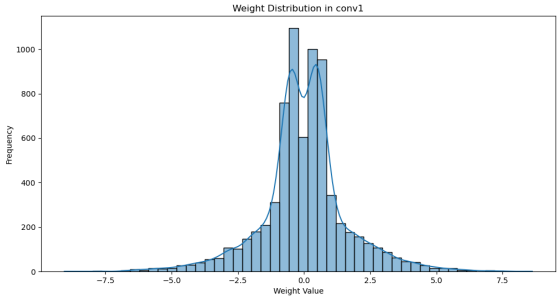

Figure 11: Weight distribution at first layer characteristic of Hard-WTA competition with Grossberg Instar rule.

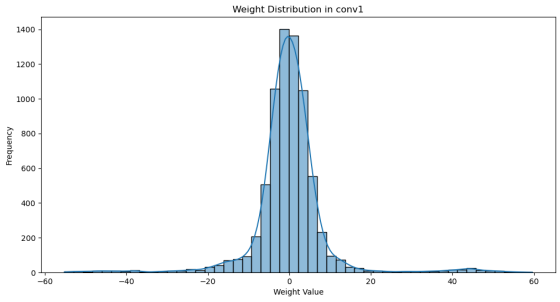

Figure 12: Weight distribution at first layer characteristic of Hard-WTA with BCM learning rule.

4.3 Configurations Qualitative Results

This section emphasises the most relevant information gleaned from our visualisation tools, highlighting the differences in hierarchical representations learnt by our 3-layer configurations.

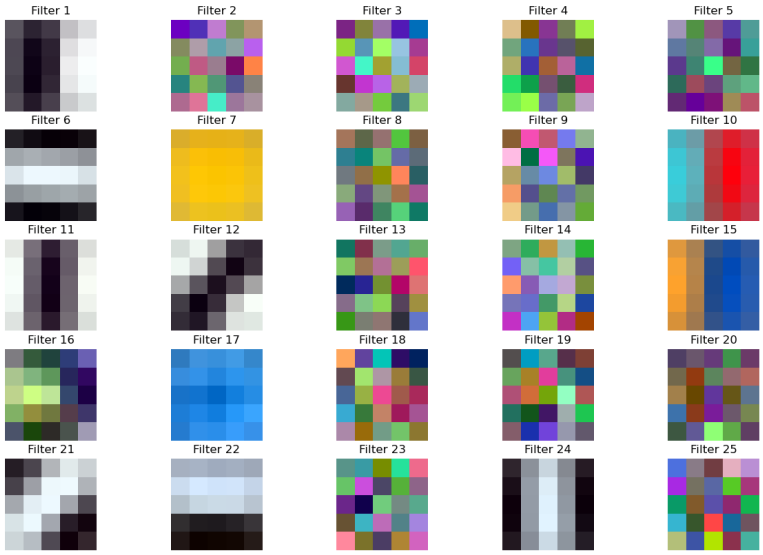

Figure 13: Plots of filters from first 25 neurons directly plotted for first layer of the **SoftHebb-Surround/HardWTA/Cos-Instar** configuration.

Figures 13 and 14 showcase the direct filters learnt at the first layer for the highest-accuracy experiments (`SoftHebb-Surr/HardWTA/Cos-Instar` and `SoftHebb-SoftWTA-Instar`), presented as 5x5 grids where each square represents a different filter or neuron, labelled from *Filter 1* to *Filter 25*. The remaining filters or neurons results per layers were also presented with this structure.

The `SoftHebb-Surr/HardWTA/Cos-Instar` configuration yielded kernels with edges or colour combinations (Filters 12 and 15 respectively in Figure 13), and the `SoftHebb-SoftWTA-Instar` configuration primarily strengthened or diminished initial weight intensities (Filters 11 and 20 respectively in Figure 14), with occasional colour-dominant kernels.

Although there is no single criterion for optimal representations since they depend on each dataset, learning rule, and architecture, we generally expected robust representations in the first convolutional layer will merge colour and edges, resulting in distributed representations across various filters. Similar results were obtained with a backpropagation model (Appendix A.1).

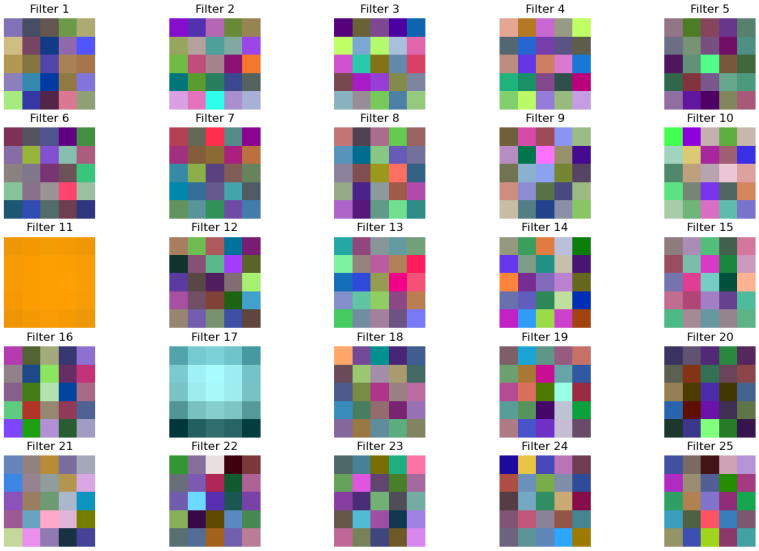

Figure 14: Plots of filters from first 25 neurons directly plotted for first layer of the **SoftHebb-SoftWTA-Instar** configuration.

The direct filter plots focused on the patterns learnt by the filters. The filters in the first convolutional layer of configurations with Hard-WTA and Soft-WTA competition, as seen by comparing Figures 13 and 14 with Figure 22, matched the representations in the receptive fields. However in deeper layers (all layers after the first convolution), as seen by comparing Figures 17 to Figure 21, the direct plots were only capable of identifying patterns like centre-surround kernels or repeating patterns. It could not capture the representations to which neurons were responsive, as the mean across channels reduces these direct kernel plots to patterns.

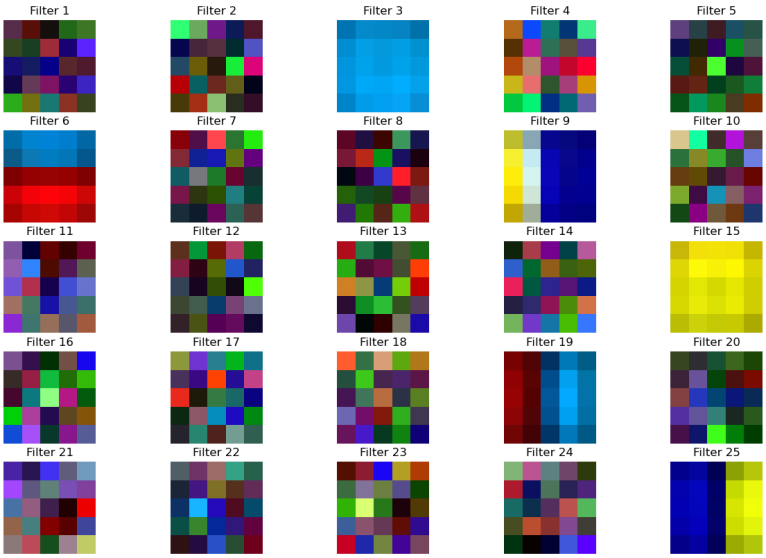

Figure 15: Plots of filters from first 25 neurons directly plotted for first layer of **Dale_HardHebb-Surr/HardWTA/Cos-Instar** configuration.

Figures 15 and 16 highlight the unique filters for `Dale_SoftHebb-SoftWTA-Instar` and `SoftHebb-HardWTA/Cos-BCM` configurations that set them apart from other configurations at the first layer. For the Dale network operating under biological constraints, specialised colour filters emerged (Filter 25 in Figure 15). Models trained using the BCM rule started to combine colours and edge orientations together in its filters (Filter 2 in Figure 16).

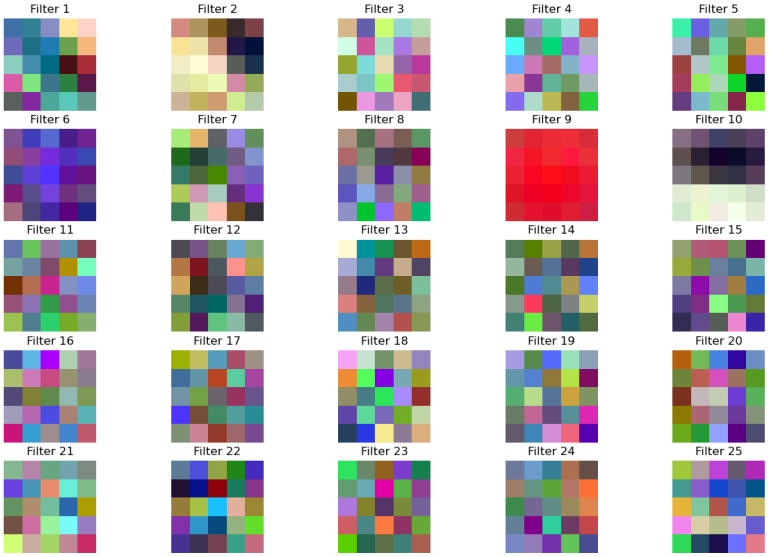

Figure 16: Plots of filters from first 25 neurons directly plotted for first layer of the **SoftHebb-HardWTA/Cos-BCM** configuration.

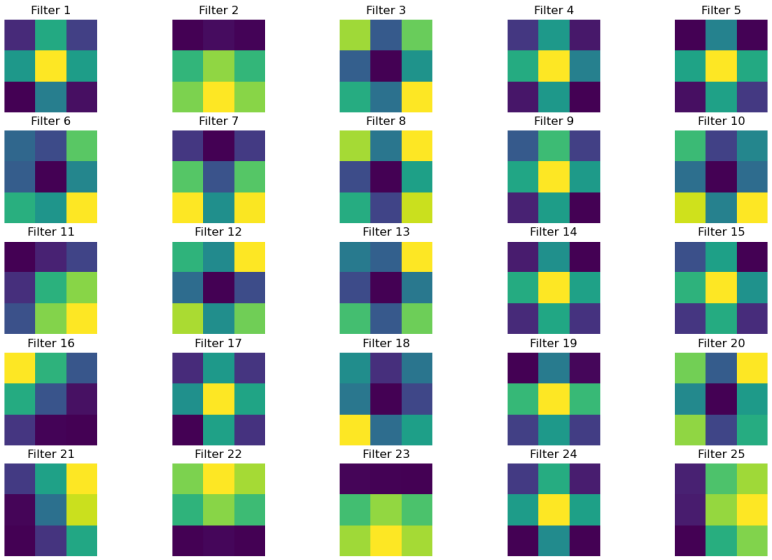

Figure 17: Filter patterns from first 25 neurons for last layer of the **Depthwise_SoftHebb-Surround/HardWTA/Cos-Instar** configuration

Figures 17 and 18 present the filter patterns learnt at the last layer for `Depthwise_SoftHebb-Surround/HardWTA/Cos-Instar` and `Dale_SoftHebb-Surround/HardWTA/Cos-Instar` configurations. The Depthwise model produced different kernel patterns, with a dominance of centre-surround kernels (Filters 11-13 and 4-5 respectively in Figure 17), while the Dale Principle experiment formed exclusively centre-surround kernels.

This natural emergence of centre surround kernels with both `Depthwise_SoftHebb-Surround/HardWTA/Cos-Instar` and `Dale_SoftHebb-Surround/HardWTA/Cos-Instar` reinforces the biological plausibility of combining Hebbian learning and depthwise convolutions. Both present different approaches to learning in deeper layers, either presenting a diverse configuration of patterns or a dominance of biologically realistic centre-surround patterns.

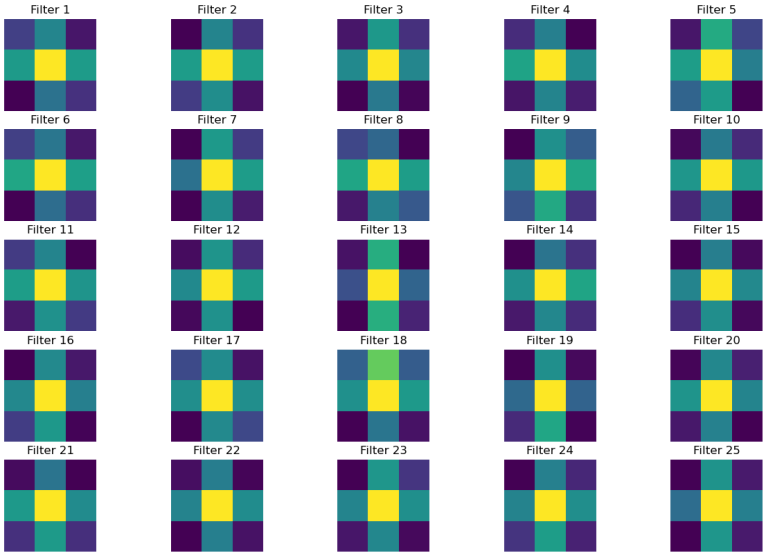

Figure 18: Filter patterns from first 25 neurons for last layer of the **Dale_SoftHebb-Surround/HardWTA/Cos-Instar** configuration.

The class clustering and separation focused on the learnt final representations from a neural network of classes in the dataset. A successfully learnt representation will cluster instances of a same class together and will separate them from instances of other classes. Classes which are similar to each other, such as species of a same animal, will be clustered together given their similarity in features. Colours are used to label classes and help identify the formation of clusters.

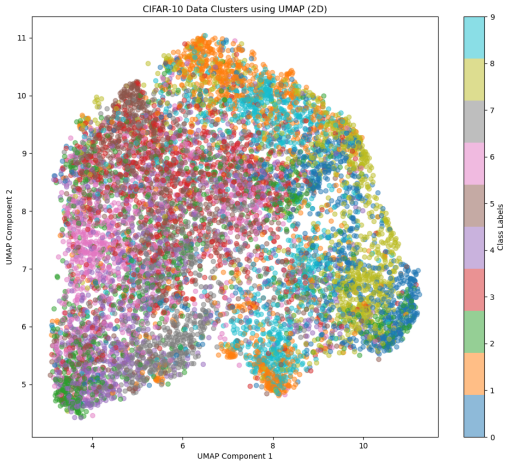

Figure 19: Plots of class clustering/separation of the **SoftHebb-Surround/HardWTA/Cos-Instar** configuration.

Figures 19 and 20 display the class clustering obtained from our highest accuracy `SoftHebb-Surr/HardWTA/Cos-Instar` and `SoftHebb-SoftWTA-Instar` configurations. Each class formed its own cluster or different clusters, and attempts at separation between clusters was found in both models, with `SoftHebb-SoftWTA-Instar` displaying more concentrated clusters. These findings suggest hierarchical learning is occurring, with the model recognising features shared by instances within the same classes and distinguishing them from those in different classes.

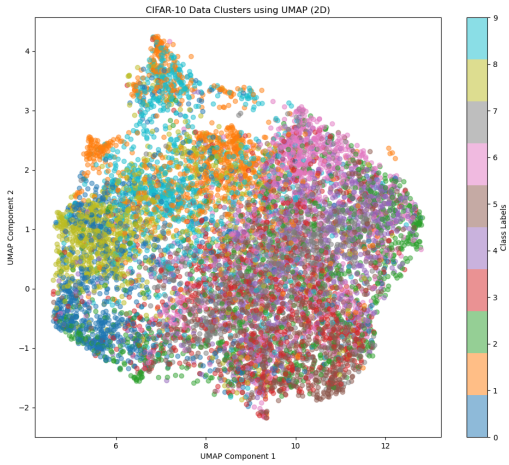

Figure 20: Plots of class clustering/separation of the **SoftHebb-SoftWTA-Instar** configuration.

Our new plots, LTP/LTD per pixel of a kernel, indicated whether the mean of a pixel in a filter grows or decreases in strength as well as when weights have converged and no further updates will change their behaviour. The stored change in weights in the temporary memory buffer was used to determine the values of LTP/LTD. Pixels shaded in green denoted Long-Term Potentiation (LTP) or an increase in weight strength, while those shaded in red represented Long-Term Depression (LTD) or a decrease in weight strength. The neutral beige tone meant no change in weight strength, indicating weight convergence.

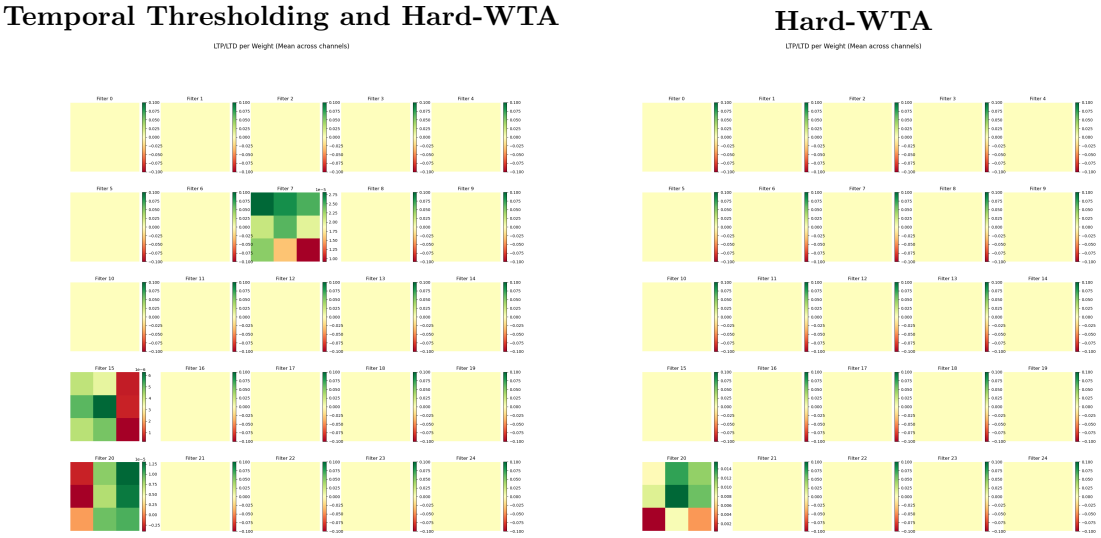

Figure 21: New LTP/LTD plots for first 25 neurons of last layer of **SoftHebb-Temp/HardWTA/Cos-Instar** and **SoftHebb-HardWTA/Cos-Instar** configurations.

Figure 21 illustrates an example of our novel visualisation method to understand convergence of Hebbian learning in the `SoftHebb-Temp/HardWTA/Cos-Instar` and `SoftHebb-HardWTA/Cos-Instar` configurations. Temporal selectivity permitted more neurons to update over extended timesteps, whereas in the absence of this competition mechanisms, most weights had converged.

Receptive fields captured the inputs which maximally activate neurons in a specific layer. The expected representations in the first layer should be simple patterns such as edges and colours which match the direct visualisations from Figure 13 and 14, and increasingly abstract shapes in deeper layers (Figure 24). This hierarchical process was seen in the receptive fields of the same configurations at the first and last layers in Figures 22 and 24.

The deeper layers' receptive fields lacked recognisable object shapes (anatomical features of a cat as an example), as we processed 32x32 pixel images through small 3-layer models. This constraint resulted in abstract feature extraction rather than object-specific features from classes.

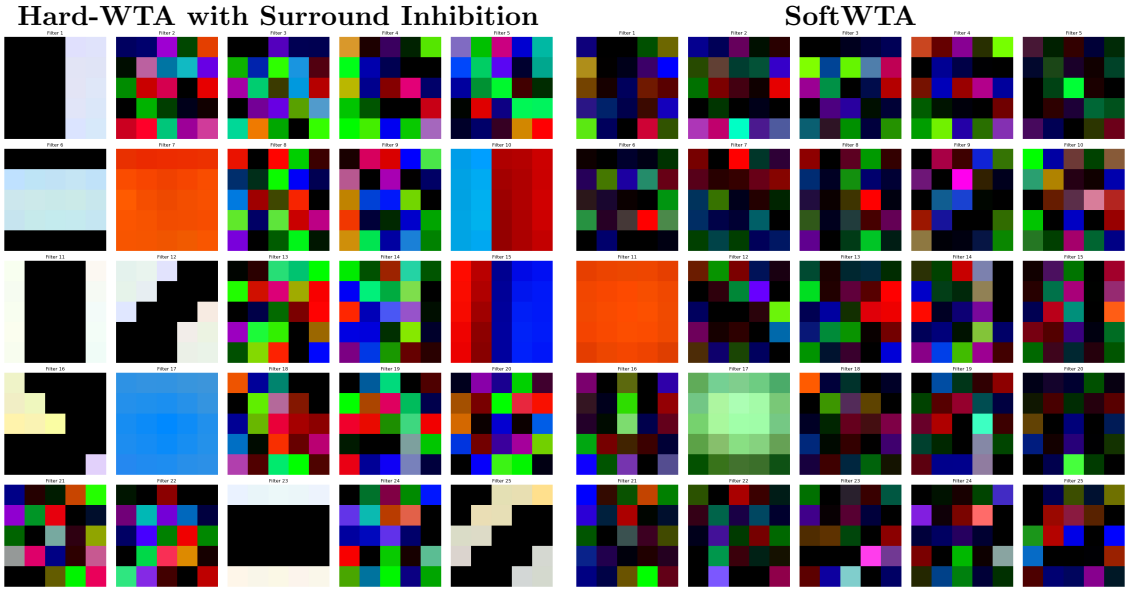

Figure 22: Plots of receptive fields for first 25 neurons of first layer of **SoftHebb-Surround/HardWTA/Cos-Instar** and **SoftHebb-SoftWTA-Instar** configurations.

Figure 22 displays the receptive fields for neurons at the first convolutional layer for our `SoftHebb-Surround/HardWTA/Cos-Instar` and `SoftHebb-SoftWTA-Instar` configurations. Edges with orientations, selective uniform colours or opposing colours (orange-blue) activated neurons trained with Hard-WTA competition, matching the direct visualisation of filters at this layer from Figures 13-14. On the other hand, Soft-WTA retained the initial random patterns from initialisation, and adjusted the intensity of these patterns, causing receptive fields at the first layer to focus on distributed representations.

Figure 23 displays the results of `SoftHebb-None-Instar` and `SoftHebb-Stats/HardWTA/Cos-Instar` configurations. Without competition, neither meaningful edge orientations nor distributed patterns were captured by the receptive fields, highlighting the importance of competition mechanisms to capture discriminative features. Statistical thresholding allowed all neurons to learn different information, proving its homeostatic capabilities. However, the neurons could not form distinct shapes and only colour sensitive receptive fields, indicating hard-WTA competition was not functioning strongly and homeostasis was instead dominating competition.

Figure 24 displays the receptive fields at the deepest layer of our best configurations, `SoftHebb-Surround/HardWTA/Cos-Instar` and `SoftHebb-SoftWTA-Instar`. Both displayed abstract shapes and patterns, which combined different colours with increasing complexity. The shapes of patterns with Surround Lateral Inhibition were smoother when compared to both `SoftHebb-SoftWTA-Instar` and its counterpart without this competition mechanism in Figure 25, indicating lateral inhibition was providing an additional smoothing property on the learnt representations, which was not captured in previous literature.

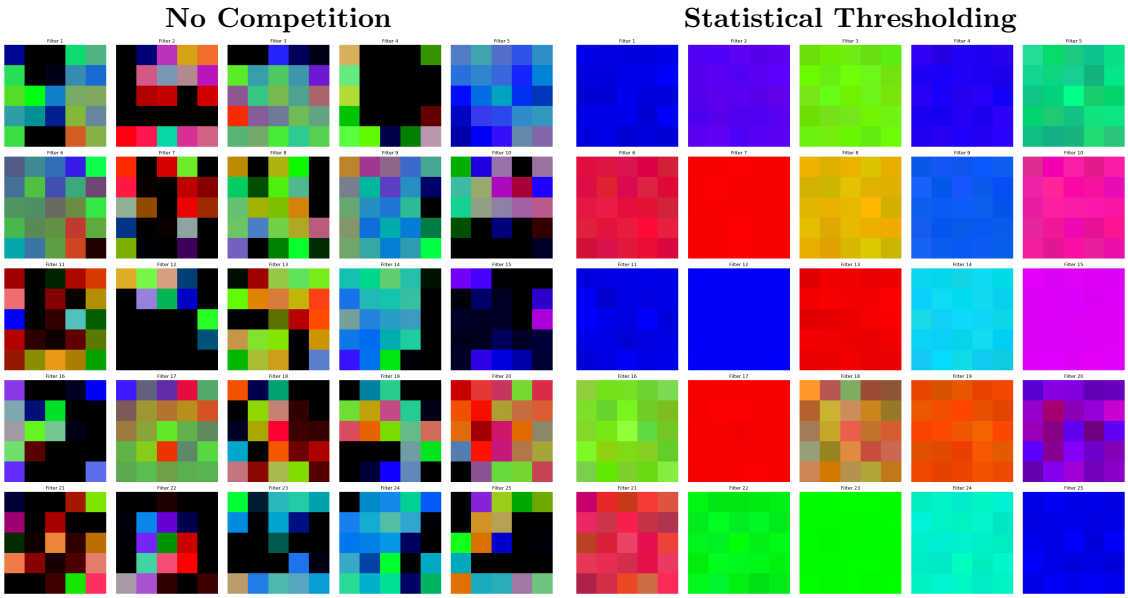

Figure 23: Plots of receptive fields for first 25 neurons of first layer of **SoftHebb-None-Instar** and **SoftHebb-Stats/HardWTA/Cos-Instar** configurations.

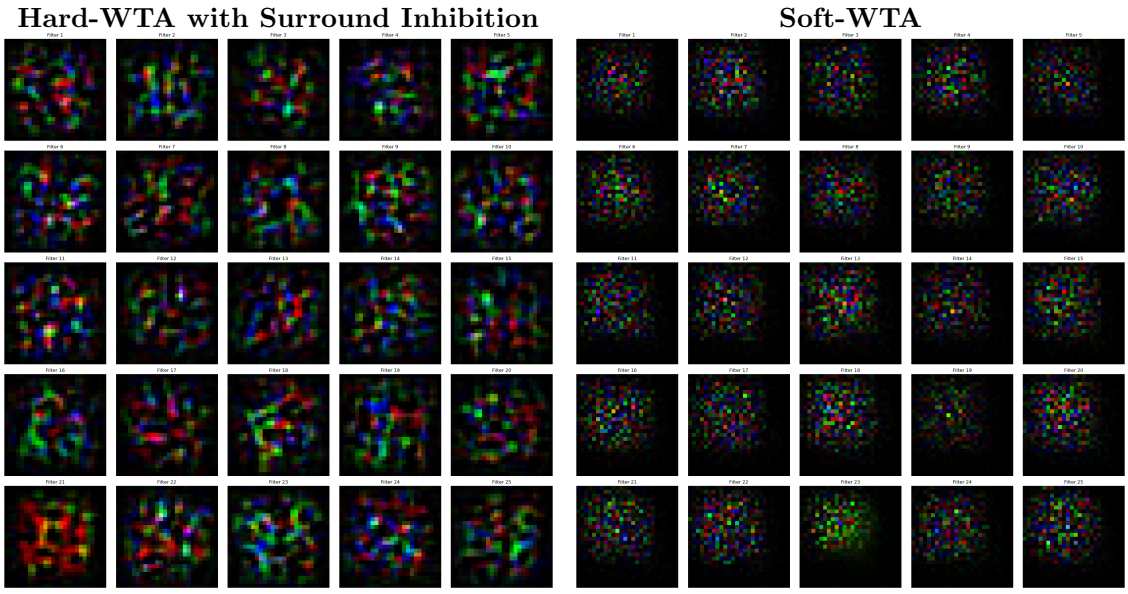

Figure 24: Plots of receptive fields for first 25 neurons of last layer of **SoftHebb-Surround/HardWTA/Cos-Instar** and **SoftHebb-SoftWTA-Instar** configurations.

Figure 25 presents insightful results for the receptive fields at the deepest layer of `SoftHebb-HardWTA/Cos-Instar` and `SoftHebb-None-Instar` configurations. The receptive fields of `SoftHebb-HardWTA/Cos-Instar` were comparable to `SoftHebb-SoftWTA-Instar`, with distributed representations of abstract patterns with diverse colours. The receptive fields without any form of competition mechanisms presented repeated patterns of single colours, which indicated unsuccessful representational learning.

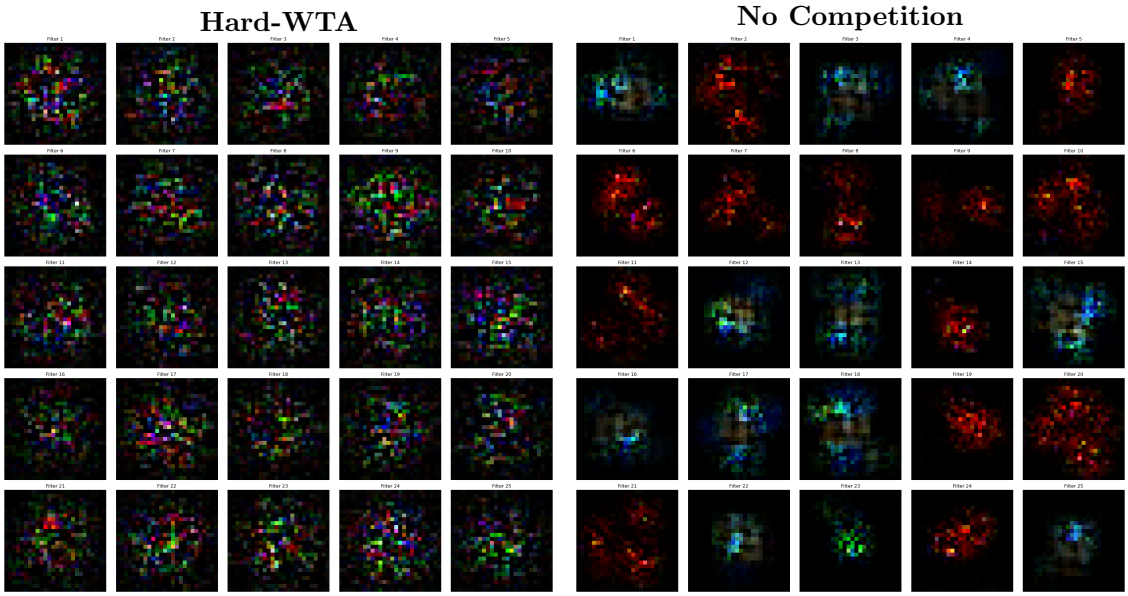

Figure 25: Plots of receptive fields for first 25 neurons of last layer of **SoftHebb-HardWTA/Cos-Instar** and **SoftHebb-None-Instar** configurations.

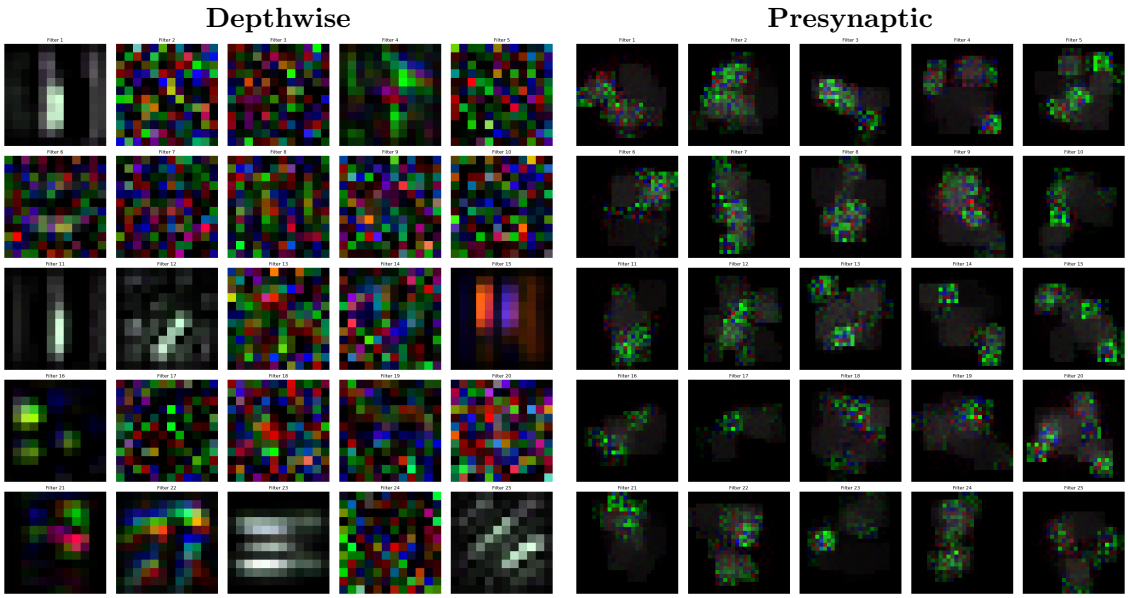

Figure 26: Plots of receptive fields for first 25 neurons of second separable layer of **Depthwise_SoftHebb-HardWTA/Cos-Instar** and last layer of **SoftHebb-Presynaptic/HardWTA/Cos-Instar** configurations.

Figure 26 illustrates distinctive results obtained from the receptive fields of the second layer of the depthwise architecture with `Depthwise_SoftHebb-HardWTA/Cos-Instar` and the unique results obtained at the last layer with `SoftHebb-Presynaptic/HardWTA/Cos-Instar`. For the depthwise configuration, the receptive fields of the depthwise convolution, which focuses solely on spatial features, correctly displayed spatial information, with a focus on edges and colour combinations even at deeper layers, as well as the emergence of more complex patterns and shapes. For the presynaptic competition, unique and diverse clusters dominated by green intensities were captured by neurons.

5 Discussion

5.1 Analysis of results

The experimental results demonstrate varying efficacy across different Hebbian learning approaches in CNNs. `SoftHebb-SoftWTA-Instar` emerges as a superior method, achieving over 70% accuracy across all benchmarks. The weight distribution in Figure 11 closely approximates the original normal distribution 8, suggesting stability and distributed representations within the network. Notably, `SoftHebb-SoftWTA-Instar` exhibits robust class clustering and separation, indicative of effective hierarchical learning in Figure 20. This performance can be attributed to its capacity to form incrementally complex and abstract representations, as demonstrated by the receptive fields of neurons at each layer in Figure 24.

The HardWTA mode in `Lagani-HardWTA/Cos-Instar`, implemented with the original Lagani Architecture and cosine similarity activation during both forward propagation and weight updates as proposed by Amato et al. (2019), achieves an accuracy exceeding 55%. This approach forms basic representations in its receptive fields, progressing from edges and colours in earlier layers as seen in Figure 13 to simple shapes and diverse patterns in deeper layers. While class clustering is observed in Figure, it demonstrates limited efficacy in separating similar classes, indicating basic hierarchical learning. The simple representations can be seen in Appendix A.1 Figure 30.

Further experiments were conducted to assess the impact of different architectures on the performance of hard-WTA competition in `Lagani_Deep-HardWTA/Cos-Instar`. Notably, the original 4-layer architecture by (Amato et al. 2019) attained 53% accuracy, supporting the reported performance decline with the addition of deeper layers. Significant improvements were realised when we adopted an architecture similar to SoftHebb without padding, denominated as `HardHebb` model. This modified approach yielded substantial accuracy improvements, surpassing 65% accuracy.

A critical result emerged from our implementation of HardWTA with cosine similarity activation function and the exact SoftHebb Architecture. This `SoftHebb-HardWTA/Cos-Instar` configuration achieved state-of-the-art performance with an impressive 72% accuracy, representing the highest performance attained in the deepest network yet for a fully Hard-WTA Hebbian-CNN model. The hierarchical representations formed are comparable to those of SoftHebb, with each layer exhibiting abstract and distributed learned representations as seen in Figure 25.

Given its superior performance, the SoftHebb architecture was adopted as the default CNN model for subsequent experiments. With the absence of competition in `SoftHebb-None-Instar`, the network's performance deteriorates significantly, reaching only 19% accuracy due to its inability to extract meaningful features. This is demonstrated by a focus on colours in the receptive fields of neurons across all layers as seen in Figure 23 and 25, representational collapse in class clusters, and signs of weight convergence as indicated by a negatively-skewed distribution with a sharp peak.

The BCM learning rule in `SoftHebb-HardWTA/Cos-BCM` also achieved state-of-the-art performance with 71% accuracy but behaved vastly differently than its Grossberg Instar counterpart. While the deeper representations remain similar, the initial layer displays filters which combine distributed representations with edges. The distribution of weights are also significantly distinct with a sparse distribution as seen in in Figure 12, where the mean is close to zero but presents a large standard deviation. This suggests sparse connections could be removed for further optimisation and reduction in parameters.

Novel temporal and statistical competition modes were evaluated alongside Hard-WTA competition using the SoftHebb architecture. The temporal competition in `SoftHebb-Temp/HardWTA/Cos-Instar` obtained comparable results to `SoftHebb-HardWTA/Cos-Instar`, with 71% accuracy.

Weights had also not all converged yet, as seen by Figure 21. Notably, the absence of WTA specialisation led to representational collapse in clusters and weight convergence, highlighting its necessity.

The statistical threshold method in `SoftHebb-Stats/HardWTA/Cos-Instar`, when evaluated with WTA competition, yielded an accuracy of 60%. Interestingly, this approach demonstrated a stronger response to colours rather than shapes and patterns as seen in Figure 23. Despite the simplicity of these filters, the model successfully separated class clusters.

Various iterations of presynaptic competition were tested in `SoftHebb-Presynaptic/HardWTA/Cos-Instar`, with relative success shown only in the input channels competition dimension. Other versions exhibited representational collapse, characterised by minor changes to weights in the first layer but significant alterations in deeper layer patterns. Spatial and output channel-based competition resulted in repeated simple patterns and shapes in the receptive fields and filters, showcasing convergence of weights and redundant representations. These receptive fields were dominated by either green colour sensitivity as seen in Figure 26 or exclusively RGB colour dominance in the case of output and spatial competition modes, while global presynaptic competition produced noisy receptive fields. Examples of these representations can be seen in Appendix A.1 Figure 32.

The simple surround inhibition kernel between neurons of a same filter in `SoftHebb-Surround/HardWTA/Cos-Instar` provided a refinement over `SoftHebb-HardWTA/Cos-Instar`, enhancing edges and smoothing uniform features achieving a light improvement with an accuracy of 73%, a new milestone in Hard-WTA competition. The receptive fields of deeper layers are smoother abstract shapes than the previous best hard-WTA configuration, which could permit increased generalisability, as seen in Figure 24. Given this improvement, it is used in all subsequent Hard-WTA configurations.

A depthwise architecture equivalent to the SoftHebb model with `Depthwise_SoftHebb-Surr/HardWTA/Cos-Instar`, achieved near state-of-the-art performance with 69% accuracy while reducing parameters by a factor of 6.6 (from 5,893,162 to 883,978 parameters). Visualisation of the receptive fields in depthwise convolutions provided unique insights into the learning process of this novel architecture. The second layer demonstrated a larger focus on edges and shapes, with first-layer filters producing edges also generating edges in their counterparts in the following layer as seen in Figure 26, while noisier distributed representations in the first layer led to similar patterns in the subsequent filter counterparts.

The network partially formed by residual blocks of depthwise separable convolutions in `Residual_SoftHebb-Surr/HardWTA/Cos-Instar`, similar to MobileNet, performed comparably to `Depthwise_SoftHebb-Surr/HardWTA/Cos-Instar` in terms of class separation and weight distribution across layers. A notable difference was observed in the features detected by the receptive fields of the second layer, where residual blocks focused on specific spots of colour rather than edges and lines. This difference can be noted in Appendix A.1 Figure 31.

Lastly, a depthwise network fully respecting Dale's principle with only excitatory synapses was evaluated to enhance the biological plausibility of the model in `Dale_SoftHebb-Surr/HardWTA/Cos-Instar`. The SoftHebb architecture with padding produced inferior results, forming only basic filters of colours and edges. In contrast, the HardHebb architecture without padding improved performance, allowing for the formation of edge and colour filters.

Significantly, in the depthwise convolutions of the Dale variant, most filters exhibited centre-surround patterns as seen in Figure 18, further accentuating the biological plausibility of this network. This difference may help explain the biological implausibility of padding, as neurons cannot create artificial regional extensions to inputs from beyond their receptive fields, further proving the biological plausibility of this variant.

Comparative analysis of backpropagation using an identical 3-layer architecture to SoftHebb in `SoftHebb-Backpropagation` revealed comparable performance outcomes, irrespective of whether the weight initialisation replicated our Hebbian models or employed the default PyTorch initialisation. A notable observation emerged when utilising our initialisation scheme as the initial filter patterns remained largely unaltered, with only subtle, imperceptible modifications occurring. In contrast, the default weight initialisation led to the emergence of filters in the first layer that combined edges and colours, while deeper layers exhibited more significant alterations to the original patterns, occasionally producing centre-surround kernels. These results can be found in Appendix A.1 Figures 27, 28, and 29.

Accuracy metrics for both initialisation methods converged at approximately 70%, highlighting the competitive efficacy of Hebbian learning in shallow networks. However, it is important to note that the strengths of backpropagation become more pronounced with the incorporation of additional deep layers and extended training epochs. These augmentations facilitate further optimisation, surpassing the performance ceiling of Hebbian learning approaches in more complex network architectures.

5.2 Objectives Evaluation

This research successfully addressed the established objectives, contributing significantly to the field of Hebbian learning in Convolutional Neural Networks and benefiting AI researchers, neuroscientists and future industry applications. The outcomes for each objective are as follows:

The **first objective** of developing a baseline model with a simple Hebbian learning rule, utilising datasets and parameters consistent with existing research for fair comparisons was successfully attained. This achievement facilitated a fundamental understanding of basic Hebbian learning and its integration into CNNs, while also enabling rigorous examination of previous implementations.

The **second objective** of implementing state-of-the-art research and ensuring similar results was completed despite encountering unanticipated complexities that extended the development timeline. It was the most essential step, as through understanding these implementations and their inner workings we successfully replicated results from literature. To complete this objective, some implementation details were extracted from the code as they did not appear in the published papers, with the aim of facilitating future implementation.

The **third objective** of understanding limitations in these implementations and researching extensions to enhance performance was mostly completed. As this is a novel field inside Artificial Intelligence, research regarding Hebbian-CNN literature was limited to specific papers, and additional interdisciplinary insights from biological-inspired methodologies and neuroscience-based research were used to find some extensions to state-of-the-art methods. Optimisation of these methods was a major focus, from a computational and structural perspective.

Finally, the **last objective** of comprehensive experiments and comparative analysis between experiments was performed successfully. This was accomplished with correct evaluation metrics and by devising a suite of visualisation tools new to this field and to the machine learning community. These new tools allow an interpretable and explainable understanding of learning processes in neural networks, with a focus on Hebbian learning but applicable to backpropagation as well.

5.3 Key Findings

The key finding in this project has been the success of combining Hard-WTA competition and Hebbian learning to achieve state-of-the-art performance through an architecture with the

deepest number of Hebbian layers in literature. The scaling factor of number of filters per layer from SoftHebb also has a large impact on the performance of Hard-WTA competition, undocumented in literature before this project.

Extensions to competition, learning rules and lateral interactions were evaluated, with noted success in the incorporation of a surround modulation kernel to simulate lateral inhibition, improving by 1% accuracy of our best Hard-WTA configuration and creating smoother representations which can aid with generalisability. Temporal competition alongside hard-WTA allowed updates to occur in deeper layers during longer timesteps before weight convergence, indicating more selectivity in the updates of neurons, while statistical thresholds provided a type of indirect homeostasis, albeit an overly aggressive form.

BCM learning rule was integrated into Hebbian-CNN and behaved differently from all other experiments. Rather than retaining the basic form of a Gaussian distribution in its weights, it creates a very sparse distribution with most values close to 0 and incrementally larger heavy tails. Even with sparse weights, it can form efficient representations which combine edge and colour information and achieve state-of-the-art performance, indicating its efficiency and adequacy in neural networks.

The first implementation of Hebbian Depthwise Separable Convolutions in this field substantially decreases the parameter count by a factor of 6 while maintaining state-of-the-art performance. This reduction in parameters from a computational standpoint can alleviate the increased number of filters required to achieve high precision, while increasing its biological plausibility. This architecture is beneficial for both Soft and Hard-WTA competition.

Residual blocks increased the depth of the network by including additional pointwise convolutions, achieving high accuracy on a deeper network, reinforcing the importance of additional connections such as skip connections to allow deeper training in Hebbian networks. A Depthwise Separable Convolutional Hebbian network trained following Dale's principle naturally produces a type of centre-surround kernels in deeper layers, supporting the biological plausibility of this convolution with Hebbian learning and strengthening the research of Babaiee et al. (2024).

These findings exceeded our initial expectations for the objectives, yielding outstanding outcomes through a learning dynamic that is both computationally efficient and biologically plausible, capable of competing with the widely-used backpropagation algorithm, offering a new optimised training paradigm for Artificial Intelligence.

5.4 Answer to Research Question

The main question for this research project was: **Can Hebbian learning algorithms achieve representational learning comparable to backpropagation while maintaining biological plausibility?**

This question can be answered affirmatively with many different Hebbian configurations under certain constraints. In shallow networks of 3 layers and under training time constraints, Hebbian Learning in CNN with competition mechanisms can achieve or surpass backpropagation at a faster rate for both training and test sets. This result is encouraging for the application of this biologically plausible learning rule in more complex environments and tasks, and may possess stronger generalisation capabilities given its unsupervised foundation.

Further sub-questions where also posed:

- How do different Hebbian learning rules affect the network's ability to form hierarchical representations?

  This question can be answered with the differences in behaviour found between the

Grossberg Instar and the BCM learning rule under the same conditions, where the first rule supports a broader spread of synaptic strengths while BCM learning rule focuses on selectivity through sparsity.

- What role does neuronal competition play in Hebbian learning?

  Competition between neurons is crucial for creating meaningful representations which extract discriminative features from data to achieve high performance. Hard-WTA competition can attain comparable performance to Soft-WTA competition, and lateral mechanisms can help reinforce competition and selectivity between neurons.

6 Evaluation, Reflection and Conclusion

6.1 Evaluation and Reflection of Objectives and Findings

The objectives laid out permitted a structured approach to the project, starting from a solid foundation with a baseline and improving this starting point with state-of-the-art Hebbian-CNN research. Once results from literature could be obtained, extensions and understanding of the limitations of these methods became the intuitive step to improve on these results. Finally, carefully controlled and rigorous experiments facilitated an analysis of emergent behaviours from competition mechanisms between neurons and how they interact with different learning rules and architectures.

This bottom-up approach provided flexibility and focus at each step, and these properties permitted the successful completion of these objectives. The structured approach facilitated by our objectives provided a robust framework for the project's progression, systematically incorporating state-of-the-art Hebbian-CNN research and extending our understanding of these methods' limitations.

Regarding our main findings, we are pleased by the results obtained through our thorough analysis of hard-WTA competition. Achieving state-of-the-art performance through architectural changes is a simple yet effective modification, emphasising the importance of increasing diversity in neurons to allow correct winner selection. This demonstrates that simple yet effective changes can yield substantial improvements.

Further architectural changes to preserve more biological plausibility while reducing parameters also maintained SOTA accuracy, a surprising result alongside the natural emergence of centre-surround kernels when following Dale's Principle. This unexpected outcome highlights the potential for biologically inspired approaches to yield competitive performance.

Residual networks also achieve close to SOTA performance, achieving the best performance for the deepest Hebbian network yet. This represents a significant advancement in the depth of Hebbian networks while maintaining high accuracy.

Finally, temporal and statistical competition thresholds were designed for the first time, and can be expanded upon to improve the generalisability of the model. Pre-synaptic competition completely changes the behaviour of the model, and further experiments to integrate it with post-synaptic competition must be studied. These novel designs open new avenues for improving model generalisability and understanding the complex interactions between different types of neuronal competition.

6.2 Evaluation of Limitations

Our main limitation has been the lack of full customisation of a model layer by layer, and the control of learning rules and competition mechanisms through command arguments. Currently, all layers in a model have the same competition and learning rules applied to them. To change some of these learning mechanisms, manual changes must be made in their respective files to allow it to work. This constraint hinders the flexibility of experimental setups and the ability to fine-tune individual layers.

Another limitation has been the results obtained from the extended competition mechanisms. While temporal competition has met SOTA performance, without Hard-WTA competition it cannot form representations, indicating Hard-WTA is still the main competition mechanism. Statistical competition only creates simple representations, and further tests on homeostatic mechanisms could be beneficial. These findings suggest that while our extended mechanisms

show promise, there is still room for improvement and more comprehensive understanding of their interactions and individual contributions to model performance.

6.3 Future Work

Our goal, driven by these remarkable findings, is to write this project for publication, sharing the insights gained from merging Hebbian learning and local competition mechanisms with modern deep learning structures. This will advocate the application of Hebbian learning and its advantages to a broader scientific audience, fostering additional research in this innovative learning paradigm.

Ensuring full network customisability layer by layer and control over mechanisms through command arguments would facilitate the speed of setting up experiments and provide finer control of each layer, allowing the combination of different mechanisms in the same experiment. This enhancement would streamline the research process and enable more nuanced investigations into the interactions between different learning rules and competition mechanisms.

Evaluating these findings on different datasets and with deeper networks is paramount to understand the representational capabilities of these networks. STL-10 dataset (Coates et al. 2011) and more real-world application datasets are a good starting point. This expansion would provide a more comprehensive assessment of the model's generalisability and practical applicability.

Recurrent connections between layers could improve both performance and the biological plausibility of the network. Similarly, feedback connections from deeper layers to lower layers could improve representational formation. These additions could potentially enhance the model's ability to capture complex temporal dependencies and hierarchical relationships in the data.

To enhance the biological inspiration, distinct excitatory and inhibitory neural populations which adhere to Dale's Principle could improve formation of representation in a manner approved by neuroscientists. This approach would align our model more closely with current neuroscientific understanding, potentially yielding insights relevant to both artificial intelligence and neuroscience.

6.4 Conclusion

This project provides a competitive alternative to backpropagation through a biologically plausible learning rule, reducing the computation resources required to create meaningful representations for classification tasks. It is able to achieve state-of-the-art performance in a Hebbian-CNN architecture, with over 73% accuracy on CIFAR-10.

Hard-WTA competition was proven to be as effective as Soft-WTA competition, rivalling the performance from the benchmark set by Journé et al. (2022). New extensions including learning rules and competition mechanisms helped improved performance and laid the foundation for further research in these mechanisms.

The code offers a synthesis and optimisation of previous work, facilitating research in this intersection between neuroscience and Artificial Intelligence through an accessible platform designed with modularity and extensibility in mind. This codebase serves as a catalyst for continued improvements in both performance and biological plausibility.

In conclusion, this research contributes significantly to the field by demonstrating the viability of Hebbian learning in deep neural networks, opening new pathways for the development of more efficient and biologically inspired artificial intelligence systems. We will continue building up

from this project to improve performance and biological plausibility, and aim for applying this new technology in real-world tasks.

Ultimately, this will bridge the gap between neuroscientific principles and practical artificial intelligence applications, bringing the biological benefits of rapid training, reduced computational resources and energy efficiency minimising the effect on the ecosystem, and increased generalisability without reliance of target values in a fully unsupervised fashion.

References

Amari, S.-i. (1993), 'Backpropagation and stochastic gradient descent method', *Neurocomputing* **5**(4-5), 185–196.

Amato, G., Carrara, F., Falchi, F., Gennaro, C. & Lagani, G. (2019), Hebbian learning meets deep convolutional neural networks, *in* 'Image Analysis and Processing–ICIAP 2019: 20th International Conference, Trento, Italy, September 9–13, 2019, Proceedings, Part I 20', Springer, pp. 324–334.

Babaiee, Z., Kiasari, P. M., Rus, D. & Grosu, R. (2024), Neural echos: Depthwise convolutional filters replicate biological receptive fields, *in* 'Proceedings of the IEEE/CVF Winter Conference on Applications of Computer Vision', pp. 8216–8225.

Becker, S. & Plumbley, M. (1996), 'Unsupervised neural network learning procedures for feature extraction and classification', *Applied Intelligence* **6**(3), 185–203.

Bengio, Y. & Frasconi, P. (1993), 'Credit assignment through time: Alternatives to backpropagation', *Advances in neural information processing systems* **6**.

Bienenstock, E. L., Cooper, L. N. & Munro, P. W. (1982), 'Theory for the development of neuron selectivity: orientation specificity and binocular interaction in visual cortex', *Journal of Neuroscience* **2**(1), 32–48.

Campbell, J. et al. (2022), 'Considerations of biological plausibility in deep learning', *Cornell Undergraduate Research Journal* **1**(1), 4–12.

Cekic, M., Bakiskan, C. & Madhow, U. (2022), 'Towards robust, interpretable neural networks via hebbian/anti-hebbian learning: A software framework for training with feature-based costs', *Software Impacts* **13**, 100347.

Choe, Y. (2022), Anti-hebbian learning, *in* 'Encyclopedia of Computational Neuroscience', Springer, pp. 213–216.

Chollet, F. (2017), Xception: Deep learning with depthwise separable convolutions, *in* 'Proceedings of the IEEE conference on computer vision and pattern recognition', pp. 1251–1258.

Churhe, P. (2024), 'Unraveling deep learning: A comprehensive guide to neural networks'. Accessed: 2024-09-28.
URL: *https://prathmeshchurhe.medium.com/unraveling-deep-learning-a-comprehensive-guide-to-neural-networks-2623351da219*

Coates, A., Ng, A. & Lee, H. (2011), An analysis of single-layer networks in unsupervised feature learning, *in* 'Proceedings of the fourteenth international conference on artificial intelligence and statistics', JMLR Workshop and Conference Proceedings, pp. 215–223.

Cornford, J., Kalajdzievski, D., Leite, M., Lamarquette, A., Kullmann, D. M. & Richards, B. (2020), 'Learning to live with dale's principle: Anns with separate excitatory and inhibitory units', *bioRxiv* pp. 2020–11.

Fukushima, K. (1980), 'Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position', *Biological cybernetics* **36**(4), 193–202.

Gabbott, P. & Somogyi, P. (1986), 'Quantitative distribution of gaba-immunoreactive neurons in the visual cortex (area 17) of the cat', *Experimental brain research* **61**, 323–331.

Grossberg, S. (1976), 'Adaptive pattern classification and universal recoding: I. parallel development and coding of neural feature detectors', *Biological cybernetics* **23**(3), 121–134.

Gupta, M., Modi, S. K., Zhang, H., Lee, J. H. & Lim, J. H. (2022), 'Is bio-inspired learning better than backprop? benchmarking bio learning vs. backprop', *arXiv preprint arXiv:2212.04614* .

Hasani, H., Soleymani, M. & Aghajan, H. (2019), 'Surround modulation: A bio-inspired connectivity structure for convolutional neural networks', *Advances in neural information processing systems* **32**.

He, K., Zhang, X., Ren, S. & Sun, J. (2016), Deep residual learning for image recognition, *in* 'Proceedings of the IEEE conference on computer vision and pattern recognition', pp. 770–778.

Journé, A., Rodriguez, H. G., Guo, Q. & Moraitis, T. (2022), 'Hebbian deep learning without feedback', *arXiv preprint arXiv:2209.11883* .

Kerr, D. (2024), 'Ai brings soaring emissions for google and microsoft, a major contributor to climate change', *NPR* .
URL: *https://www.npr.org/2024/07/12/g-s1-9545/ai-brings-soaring-emissions-for-google-and-microsoft-a-major-contributor-to-climate-change*

Kheradpisheh, S. R., Ganjtabesh, M., Thorpe, S. J. & Masquelier, T. (2018), 'Stdp-based spiking deep convolutional neural networks for object recognition', *Neural Networks* **99**, 56–67.

Krizhevsky, A., Hinton, G. et al. (2009), 'Learning multiple layers of features from tiny images'.

Kucharlapati, A. (2024), 'Gradient descent: Gradient descent is an iterative optimization algorithm used to find local minima'. Accessed: 2024-09-28.
URL: *https://medium.com/@kucharlapatiaparna123/gradient-descent-gradient-descent-is-an-iterative-optimization-algorithm-used-to-find-local-minima-874d56a2ead9*

Lagani, G. (2024), 'Bio-inspired deep learning research directions', *Google Docs* .
URL: *https://docs.google.com/document/d/1mRNC0AEkrW7tcztbNuXdoz428ZCwbBW8kA2FZKZBh2o/edit*

Lagani, G., Falchi, F., Gennaro, C. & Amato, G. (2022), 'Comparing the performance of hebbian against backpropagation learning using convolutional neural networks', *Neural Computing and Applications* **34**(8), 6503–6519.

Lagani, G., Falchi, F., Gennaro, C. & Amato, G. (2023), 'Synaptic plasticity models and bio-inspired unsupervised deep learning: A survey', *arXiv preprint arXiv:2307.16236* .

Lagani, G., Gennaro, C., Fassold, H. & Amato, G. (2022), Fasthebb: Scaling hebbian training of deep neural networks to imagenet level, *in* 'International Conference on Similarity Search and Applications', Springer, pp. 251–264.

LeCun, Y., Bottou, L., Bengio, Y. & Haffner, P. (1998), 'Gradient-based learning applied to document recognition', *Proceedings of the IEEE* **86**(11), 2278–2324.

Lillicrap, T. P., Santoro, A., Marris, L., Akerman, C. J. & Hinton, G. (2020), 'Backpropagation and the brain', *Nature Reviews Neuroscience* **21**(6), 335–346.

Lindsay, G. W. (2018), 'Deep convolutional neural networks as models of the visual system: Qa'. Accessed: 2024-09-28.
URL: *https://gracewlindsay.com/2018/05/17/deep-convolutional-neural-networks-as-models-of-the-visual-system-qa/*

Magee, J. C. & Johnston, D. (1997), 'A synaptically controlled, associative signal for hebbian plasticity in hippocampal neurons', *Science* **275**(5297), 209–213.

Mayur, I. (2024), 'Simple convolutional neural network (cnn) for dummies in pytorch: A step-by-step guide'. Accessed: 2024-09-28.

URL: *https://medium.com/@myringoleMLGOD/simple-convolutional-neural-network-cnn-for-dummies-in-pytorch-a-step-by-step-guide-6f4109f6df80*

Miconi, T. (2017), 'Biologically plausible learning in recurrent neural networks reproduces neural dynamics observed during cognitive tasks', *Elife* **6**, e20899.

Miconi, T. (2021), 'Hebbian learning with gradients: Hebbian convolutional neural networks with modern deep learning frameworks', *arXiv preprint arXiv:2107.01729* .

Nowlan, S. (1989), 'Maximum likelihood competitive learning', *Advances in neural information processing systems* **2**.

Oja, E. (1982), 'Simplified neuron model as a principal component analyzer', *Journal of mathematical biology* **15**, 267–273.

Pogodin, R., Mehta, Y., Lillicrap, T. & Latham, P. E. (2021), 'Towards biologically plausible convolutional networks', *Advances in Neural Information Processing Systems* **34**, 13924–13936.

Pulvermüller, F., Tomasello, R., Henningsen-Schomers, M. R. & Wennekers, T. (2021), 'Biological constraints on neural network models of cognitive function', *Nature Reviews Neuroscience* **22**(8), 488–502.

Rojas, R. & Rojas, R. (1996), 'The backpropagation algorithm', *Neural networks: a systematic introduction* pp. 149–182.

Ruder, S. (2016), 'An overview of gradient descent optimization algorithms', *arXiv preprint arXiv:1609.04747* .

Rumelhart, D. E., Hinton, G. E. & Williams, R. J. (1986), 'Learning representations by back-propagating errors', *nature* **323**(6088), 533–536.

Rumelhart, D. E. & Zipser, D. (1985), 'Feature discovery by competitive learning', *Cognitive science* **9**(1), 75–112.

Schuman, C. D., Kulkarni, S. R., Parsa, M., Mitchell, J. P., Kay, B. et al. (2022), 'Opportunities for neuromorphic computing algorithms and applications', *Nature Computational Science* **2**(1), 10–19.

Sejnowski, T. J. & Tesauro, G. (1989), 'Building network learning algorithms from hebbian synapses', *Brain organization and memory: cells, systems, and circuits* pp. 338–355.

Shepherd, G. M. & Grillner, S. (2018), *Handbook of brain microcircuits*, Oxford University Press.

Song, Y., Lukasiewicz, T., Xu, Z. & Bogacz, R. (2020), 'Can the brain do backpropagation?—exact implementation of backpropagation in predictive coding networks', *Advances in neural information processing systems* **33**, 22566–22579.

Spoerer, C. J., McClure, P. & Kriegeskorte, N. (2017), 'Recurrent convolutional neural networks: a better model of biological object recognition', *Frontiers in psychology* **8**, 1551.

Sultonov, F., Park, J.-H., Yun, S., Lim, D.-W. & Kang, J.-M. (2022), 'Mixer u-net: An improved automatic road extraction from uav imagery', *Applied Sciences* **12**, 1953.

Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V. & Rabinovich, A. (2015), Going deeper with convolutions, *in* 'Proceedings of the IEEE conference on computer vision and pattern recognition', pp. 1–9.

Wu, J. (2017), 'Introduction to convolutional neural networks', *National Key Lab for Novel Software Technology. Nanjing University. China* **5**(23), 495.

A Appendix

A.1 Further Results

Additional results for the backpropagation models, both trained using our weight initialisation scheme or the default PyTorch scheme. Similar results to our Hebbian configurations were obtained when trained on the same initial weights, while the default Pytorch weights combined colours and shapes together more effectively. Class separation was also similar, displaying no meaningful differences with our Hebbian methodology. Figures 27-29 all belong to backpropagation experiments.

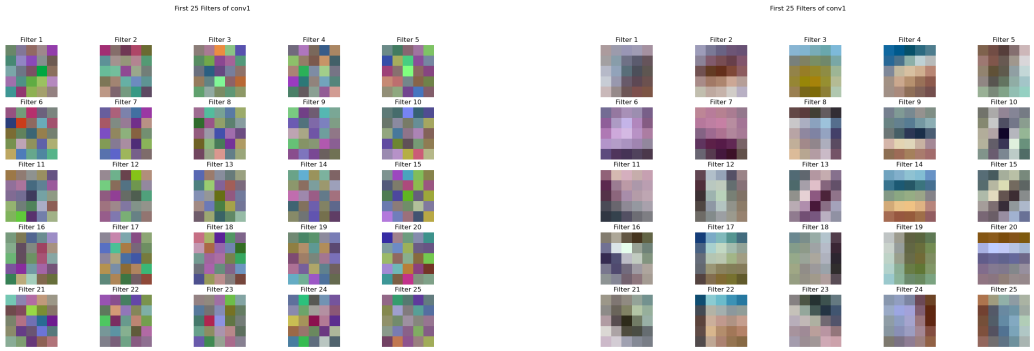

Figure 27: Direct filter plots of kernels in first layer of **SoftHebb-Backpropagation** model, using the same weight initialisation as Hebbian experiments or default Pytorch initialisation.

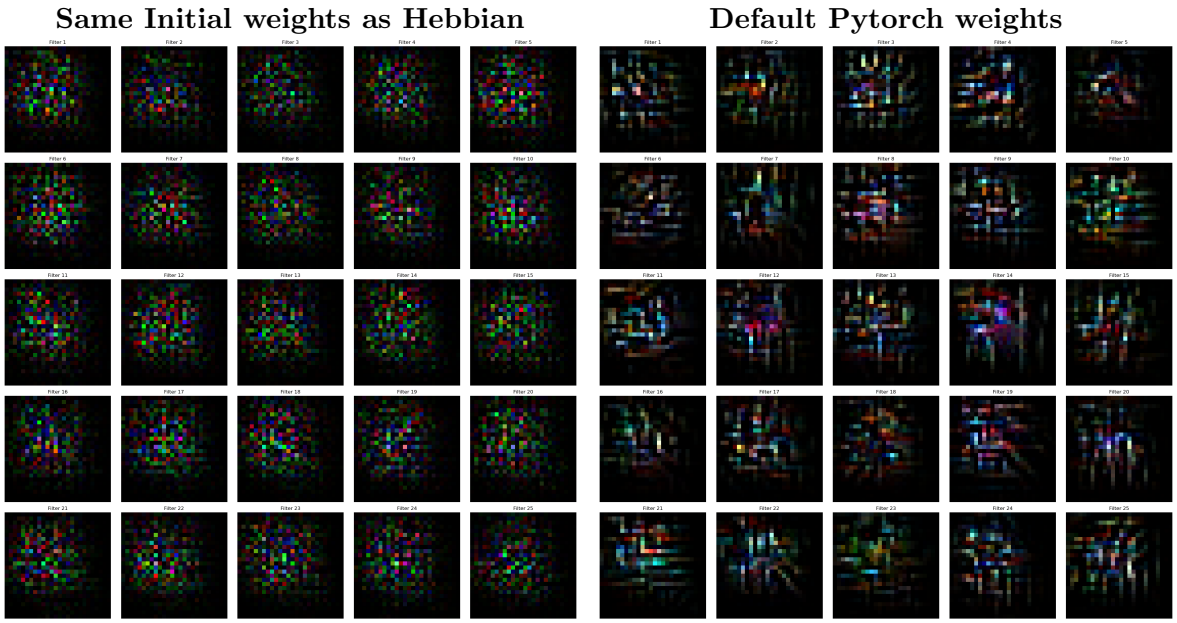

Figure 28: Receptive fields of kernels in last layer of **SoftHebb-Backpropagation** model, using the same weight initialisation as Hebbian experiments or default Pytorch initialisation.

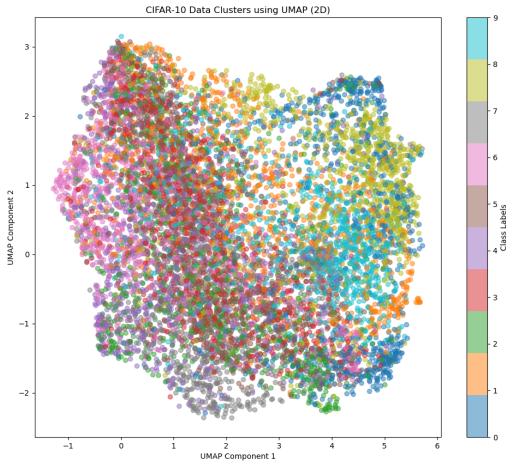

Figure 29: Plots of class clustering/separation of **SoftHebb-Backpropagation** configuration, with same initial weights as Hebbian configurations.

Results using the `Lagani-HardWTA/Cos-Instar` configuration are provided in Figure 30, to showcase how only simple patterns and shapes were formed when using a smaller architecture with smaller receptive fields and less filters per layer.

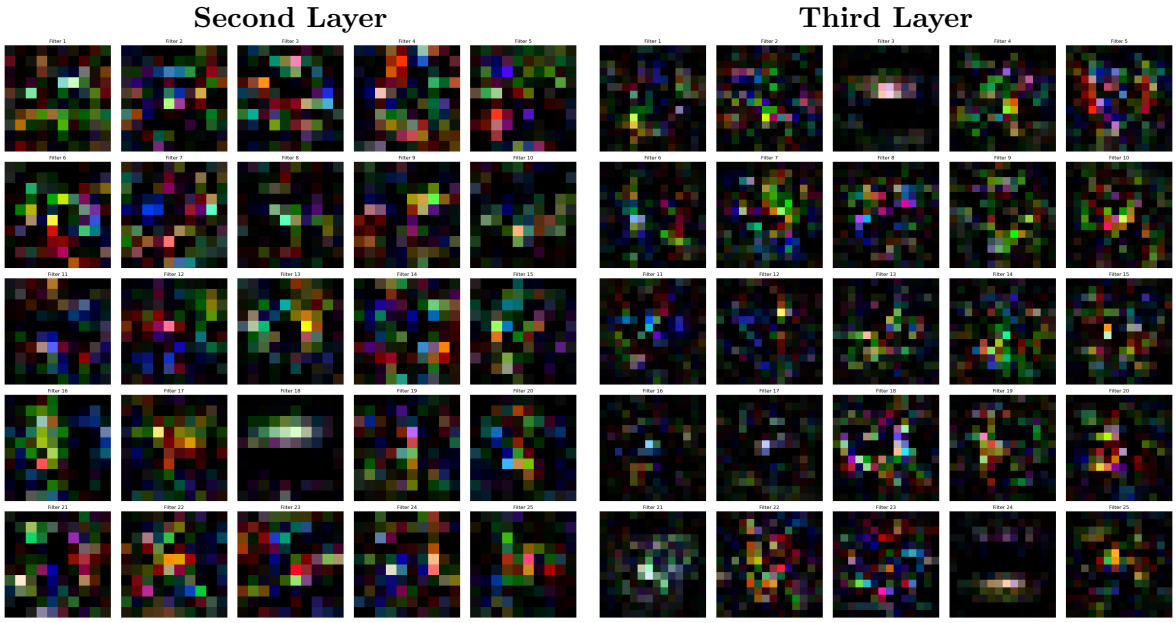

Figure 30: Receptive fields of neurons in second and last layer of **Lagani-HardWTA-Instar** configuration.

Supplementary results to aid our discussion have been provided for the residual and presynaptic configurations. Figure 31 displays how Residual depthwise convolutions focused on colours rather than edge orientations at its deeper layers, unlike the depthwise architecture. Figure 32 presents results from the presynaptic competition at a global and Output level. Global offers only noisy random representations, while output competition displays repeated focused patterns. Neither are informative for forming hierarchical representations.

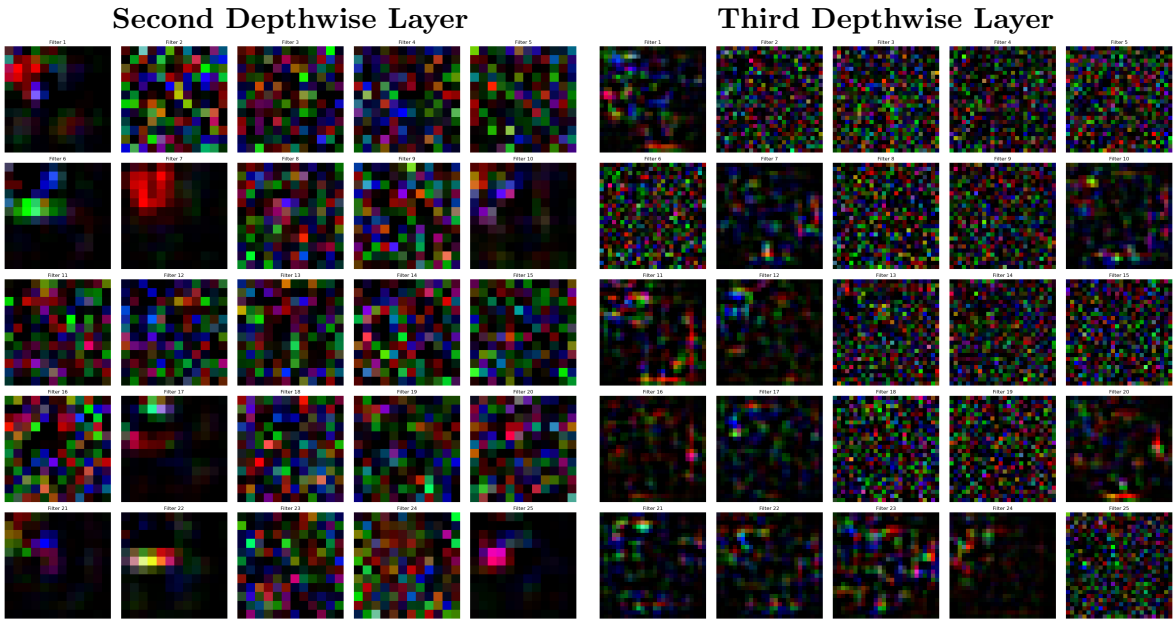

Figure 31: Receptive fields plots of kernels in second and last layer of **Residual_SoftHebb-Surr/HardWTA/Cos-Instar** configuration.

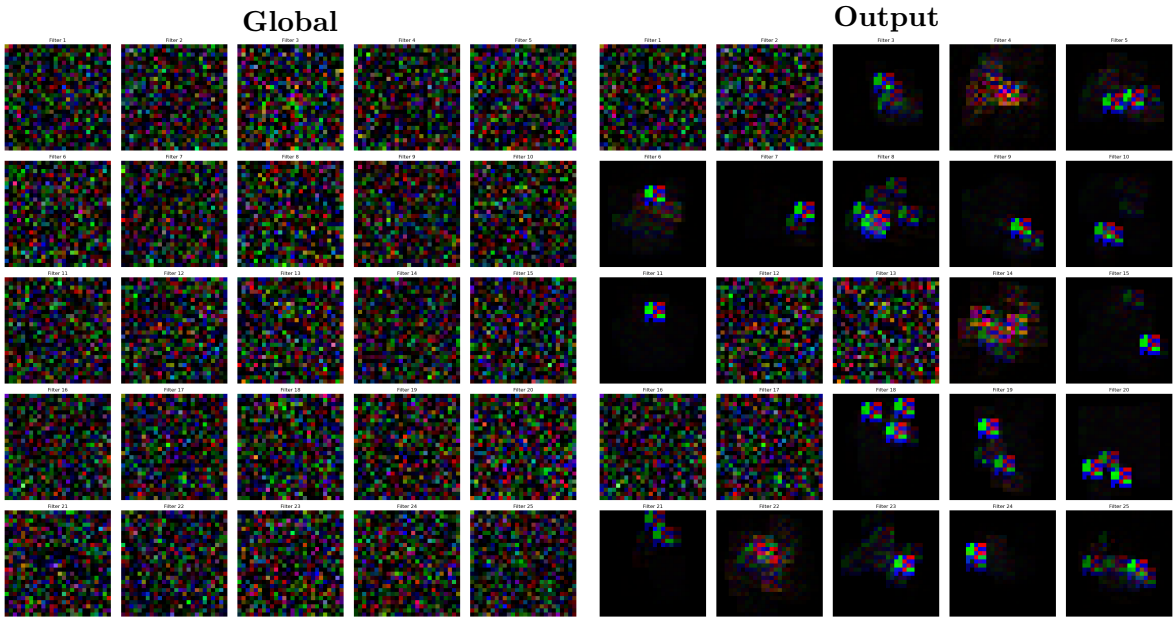

Figure 32: Receptive fields plots of kernels in last layer of configuration using Presynaptic Global or Output competition

A.2 Code Implementation

A.2.1 Data.py

```
import os
import torch
import torch.nn.functional as F
from torch.utils.data import DataLoader
from torchvision.datasets import CIFAR10, STL10
import torchvision.transforms as T
```

```python
import matplotlib.pyplot as plt
import numpy as np

import params as P


class ZCAWhitening:
    def __init__(self, epsilon=1e-1):
        self.epsilon = epsilon
        self.zca_matrix = None
        self.mean = None
        self.std = None

    def fit(self, x: torch.Tensor, transpose=True, dataset: str = "CIFAR10"):
        path = os.path.join("zca_data", dataset, f"{dataset}_zca.pt")
        os.makedirs(os.path.dirname(path), exist_ok=True)

        try:
            saved_data = torch.load(path, map_location='cpu')
            self.zca_matrix = saved_data['zca']
            self.mean = saved_data['mean']
            self.std = saved_data['std']
            print(f"Loaded pre-computed ZCA matrix for {dataset}")
        except FileNotFoundError:
            print(f"Computing ZCA matrix for {dataset}")
            if transpose and x.dim() == 4:
                x = x.permute(0, 3, 1, 2)

            x = x.reshape(x.shape[0], -1)
            self.mean = x.mean(dim=0, keepdim=True)
            self.std = x.std(dim=0, keepdim=True)
            x = (x - self.mean) / (self.std + self.epsilon)

            print(f"Data shape after reshaping: {x.shape}")
            print(
                f"Data statistics - Min: {x.min().item():.4f}, Max: {x.max().item():.4f}, Mean: {x.mean().item():.4f}, Std: {x.std().item():.4f}")

            cov = torch.mm(x.T, x) / (x.shape[0] - 1)

            # Debug print
            print(f"Covariance matrix shape: {cov.shape}")
            print(f"Covariance matrix statistics - Min: {cov.min().item():.4f}, Max: {cov.max().item():.4f}")

            # Check for NaN or inf values
            if torch.isnan(cov).any() or torch.isinf(cov).any():
                print("Warning: NaN or inf values detected in covariance matrix")
                cov = torch.where(torch.isnan(cov) | torch.isinf(cov), torch.zeros_like(cov), cov)

            if dataset == "CIFAR10":
                u, s, v = torch.svd(cov)
```

```
                inv_sqrt_s = torch.diag(1.0 / torch.sqrt(s + self.epsilon))
                self.zca_matrix = torch.mm(torch.mm(u, inv_sqrt_s), u.T)

            else:
                # Use eigendecomposition instead of SVD
                eigenvalues, eigenvectors = torch.linalg.eigh(cov)
                inv_sqrt_eigenvalues = torch.diag(1.0 / torch.sqrt(eigenvalues + self.epsilon))
                self.zca_matrix = torch.mm(torch.mm(eigenvectors, inv_sqrt_eigenvalues), eigenvectors.T)

            torch.save({'zca': self.zca_matrix, 'mean': self.mean, 'std': self.std}, path)
            print(f"Saved computed ZCA matrix for {dataset}")

    def transform(self, x: torch.Tensor):
        if self.zca_matrix is None:
            raise ValueError("ZCA matrix not computed. Call fit() first.")

        original_shape = x.shape
        x = x.reshape(x.shape[0], -1)
        x = (x - self.mean) / (self.std + self.epsilon)
        x_whitened = torch.mm(x, self.zca_matrix)
        return x_whitened.reshape(original_shape)


def whitening_zca(x: torch.Tensor, transpose=True, dataset: str = "CIFAR10"):
    zca = ZCAWhitening()
    zca.fit(x, transpose, dataset)
    return zca.transform(x)


class ZCATransformation:
    def __init__(self, zca):
        self.zca = zca

    def __call__(self, x):
        if x.dim() == 3:
            x = x.unsqueeze(0)
        x_whitened = self.zca.transform(x)
        return x_whitened.squeeze(0) if x_whitened.shape[0] == 1 else x_whitened


def visualize_zca_effect(original_data, whitened_data, num_samples=5):
    fig, axes = plt.subplots(2, num_samples, figsize=(15, 6))
    for i in range(num_samples):
        # Normalize data to [0, 1] range
        orig = original_data[i].permute(1, 2, 0)
        orig = (orig - orig.min()) / (orig.max() - orig.min())

        whit = whitened_data[i].permute(1, 2, 0)
        whit = (whit - whit.min()) / (whit.max() - whit.min())
```

```
106
107         axes[0, i].imshow(orig)
108         axes[0, i].axis('off')
109         axes[0, i].set_title('Original')
110
111         axes[1, i].imshow(whit)
112         axes[1, i].axis('off')
113         axes[1, i].set_title('Whitened')
114
115     plt.tight_layout()
116     plt.show()
117
118
119 def check_covariance(data):
120     data_flat = data.reshape(data.shape[0], -1)
121     cov_matrix = torch.cov(data_flat.T)
122     diag_mean = cov_matrix.diag().mean().item()
123     off_diag_mean = (cov_matrix - torch.diag(cov_matrix.diag())).abs
        ().mean().item()
124     print(f"Mean of diagonal elements: {diag_mean:.6f}")
125     print(f"Mean of off-diagonal elements: {off_diag_mean:.6f}")
126     print(f"Ratio of off-diagonal to diagonal: {off_diag_mean /
        diag_mean:.6f}")
127
128 def check_normalization(data):
129     data_flat = data.reshape(data.shape[0], -1)
130     mean = data_flat.mean().item()
131     std = data_flat.std().item()
132     print(f"Mean: {mean:.6f}")
133     print(f"Standard deviation: {std:.6f}")
134     print(f"Min: {data_flat.min().item():.6f}")
135     print(f"Max: {data_flat.max().item():.6f}")
136     print()
137
138
139 def get_data(dataset='cifar10', root='datasets', batch_size=32,
    num_workers=0, whiten_lvl=None):
140     trn_set, tst_set = None, None
141     if dataset == 'cifar10':
142         trn_set = CIFAR10(root=os.path.join(root, dataset), train=
            True, download=True, transform=T.ToTensor())
143         tst_set = CIFAR10(root=os.path.join(root, dataset), train=
            False, download=True, transform=T.ToTensor())
144         all_data = torch.cat([torch.tensor(trn_set.data), torch.
            tensor(tst_set.data)], dim=0)
145         all_data = all_data.float() / 255.0  # Normalize data to [0,
             1]
146
147     elif dataset == 'stl10':
148         trn_set = STL10(root=os.path.join(root, dataset), split='
            train', download=True, transform=T.ToTensor())
149         tst_set = STL10(root=os.path.join(root, dataset), split='
            test', download=True, transform=T.ToTensor())
150         all_data = torch.cat([trn_sample[0].unsqueeze(0) for
            trn_sample in trn_set] +
151                              [tst_sample[0].unsqueeze(0) for
                                 tst_sample in tst_set], dim=0)
152     else:
```

```
153         raise NotImplementedError("Dataset {} not supported.".format
                (dataset))
154
155     zca = None
156     if whiten_lvl is not None:
157         print("Data whitening")
158         zca = ZCAWhitening(epsilon=whiten_lvl)
159         zca.fit(all_data, transpose=False, dataset=dataset)
160
161         # Create a temporary loader with only ToTensor transform for
                visualization
162         if dataset=="cifar10":
163             temp_transform = T.Compose([T.Resize(32), T.ToTensor()])
164             temp_dataset = CIFAR10(root=os.path.join(root, dataset),
                    train=True, download=False,
165                                    transform=temp_transform)
166             full_transform = T.Compose([
167                 T.RandomHorizontalFlip(),
168                 T.Resize(32),
169                 T.ToTensor(),
170                 ZCATransformation(zca),
171             ])
172         else:
173             temp_transform = T.Compose([T.Resize(96), T.ToTensor()])
174             temp_dataset = STL10(root=os.path.join(root, dataset),
                    split='train', download=False,
175                                    transform=temp_transform)
176             full_transform = T.Compose([
177                 T.RandomHorizontalFlip(),
178                 T.Resize(96),
179                 T.ToTensor(),
180                 ZCATransformation(zca),
181             ])
182         # temp_loader = DataLoader(temp_dataset, batch_size=
                batch_size, shuffle=True, num_workers=num_workers)
183         # # Get a batch for visualization
184         # original_batch, _ = next(iter(temp_loader))
185         # whitened_batch = zca.transform(original_batch)
186         # print("\nVisualization of ZCA effect:")
187         # visualize_zca_effect(original_batch, whitened_batch)
188         # print("\nCovariance check:")
189         # print("Before whitening:")
190         # check_covariance(original_batch)
191         # print("\nAfter whitening:")
192         # check_covariance(whitened_batch)
193         #
194         # print("\nNormalization check:")
195         # print("Before whitening:")
196         # check_normalization(original_batch)
197         # print("\nAfter whitening:")
198         # check_normalization(whitened_batch)
199
200         # Now apply the full transform including ZCA to the datasets
201
202
203         trn_set.transform = full_transform
204         tst_set.transform = full_transform
205     else:
```

```
206         print("No ZCA")
207         if dataset == 'cifar10':
208             mean = [0.4914, 0.4822, 0.4465]
209             std = [0.2023, 0.1994, 0.2010]
210             temp_transform = T.Compose([
211                 T.RandomHorizontalFlip(),
212                 T.Resize(32),
213                 T.ToTensor(),
214                 T.Normalize(mean=mean, std=std)])
215         elif dataset == 'stl10':
216             mean = [0.4467, 0.4398, 0.4066]
217             std = [0.2603, 0.2566, 0.2713]
218             temp_transform = T.Compose([
219                 T.RandomHorizontalFlip(),
220                 T.Resize(96),
221                 T.ToTensor(),
222                 T.Normalize(mean=mean, std=std)])
223         trn_set.transform = temp_transform
224         tst_set.transform = temp_transform
225     trn_loader = DataLoader(trn_set, batch_size=batch_size, shuffle=
        False, num_workers=P.NUM_WORKERS)
226     tst_loader = DataLoader(tst_set, batch_size=batch_size, shuffle=
        False, num_workers=P.NUM_WORKERS)
227
228     return trn_loader, tst_loader, zca
229
230
231 def visualize_whitening_effect(dataset, zca, num_images=5):
232     """
233     Visualize the effect of ZCA whitening on a few sample images.
234
235     :param dataset: The dataset to sample images from
236     :param zca: The ZCAWhitening object
237     :param num_images: Number of images to visualize
238     """
239     # Get a few sample images
240     sampler = torch.utils.data.RandomSampler(dataset, num_samples=
        num_images)
241     loader = DataLoader(dataset, batch_size=1, sampler=sampler)
242
243     fig, axes = plt.subplots(num_images, 2, figsize=(10, 5 *
        num_images))
244     fig.suptitle("ZCA Whitening Effect", fontsize=16)
245
246     for i, (img, _) in enumerate(loader):
247         # Original image
248         orig_img = img.squeeze().permute(1, 2, 0).numpy()
249         axes[i, 0].imshow(orig_img)
250         axes[i, 0].set_title(f"Original Image {i + 1}")
251         axes[i, 0].axis('off')
252
253         # Whitened image
254         whitened_img = zca.transform(img).squeeze().permute(1, 2, 0)
            .numpy()
255         axes[i, 1].imshow(whitened_img)
256         axes[i, 1].set_title(f"Whitened Image {i + 1}")
257         axes[i, 1].axis('off')
258
```

```
    plt.tight_layout()
    plt.show()

# Example usage:
if __name__ == "__main__":
    # Use CIFAR-10
    cifar_trn_loader, cifar_tst_loader, cifar_zca = get_data(dataset
        ='cifar10', batch_size=64, whiten_lvl=1e-3)
    print("CIFAR-10 data loaders and ZCA whitening prepared.")



    # Use STL-10
    stl_trn_loader, stl_tst_loader, stl_zca = get_data(dataset='
        stl10', batch_size=64, whiten_lvl=1e-3)
    print("STL-10 data loaders and ZCA whitening prepared.")
```

Listing 1: Centering the Data

A.2.2 Hebb.py

```
import math

import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.nn.modules.utils import _pair
import matplotlib.pyplot as plt

# Code uses elements from {https://github.com/GabrieleLagani/
    hebbdemo} and {
# https://github.com/NeuromorphicComputing/SoftHebb}

import torch.nn.init as init

device = torch.device("cuda" if torch.cuda.is_available() else "cpu"
    )
torch.manual_seed(0)


def normalize(x, dim=None):
    nrm = (x ** 2).sum(dim=dim, keepdim=True) ** 0.5
    nrm[nrm == 0] = 1.
    return x / nrm


def symmetric_pad(x, padding):
    if padding == 0:
        return x
    return F.pad(x, (padding,) * 4, mode='reflect')


def create_sm_kernel(kernel_size=5, sigma_e=1.2, sigma_i=1.4):
    """
    Create a surround modulation kernel.
    :param kernel_size: Size of the SM kernel.
    :param sigma_e: Standard deviation for the excitatory Gaussian.
    :param sigma_i: Standard deviation for the inhibitory Gaussian.
```

```
    :return: A normalized SM kernel.
    """
    center = kernel_size // 2
    x, y = torch.meshgrid(torch.arange(kernel_size), torch.arange(
        kernel_size), indexing="ij")
    x = x.float() - center
    y = y.float() - center
    # Compute the excitatory and inhibitory Gaussians
    gaussian_e = torch.exp(-(x ** 2 + y ** 2) / (2 * sigma_e ** 2))
    gaussian_i = torch.exp(-(x ** 2 + y ** 2) / (2 * sigma_i ** 2))
    # Compute the Difference of Gaussians (DoG)
    dog = gaussian_e / (2 * math.pi * sigma_e ** 2) - gaussian_i /
        (2 * math.pi * sigma_i ** 2)
    # Normalize the DoG so that the center value is 1
    sm_kernel = dog / dog[center, center]
    return sm_kernel.unsqueeze(0).unsqueeze(0).to(device)


class HebbianConv2d(nn.Module):
    """
    A 2d convolutional layer that learns through Hebbian plasticity
    """

    MODE_HPCA = 'hpca'
    MODE_BASIC_HEBBIAN = 'basic'
    MODE_WTA = 'wta'
    MODE_SOFTWTA = 'soft'
    MODE_BCM = 'bcm'
    MODE_HARDWT = "hard"
    MODE_PRESYNAPTIC_COMPETITION = "pre"
    MODE_TEMPORAL_COMPETITION = "temp"
    MODE_ADAPTIVE_THRESHOLD = "thresh"
    MODE_ANTIHARDWT = "antihard"

    def __init__(self, in_channels, out_channels, kernel_size,
        stride=1, dilation=1, padding=0, groups=1,
                 w_nrm=False, bias=False, act=nn.Identity(),
                 mode=MODE_SOFTWTA, k=1, patchwise=True,
                 contrast=1., uniformity=False, alpha=1.,
                    wta_competition='similarity_spatial',
                 lateral_competition="combined",
                 lateral_inhibition_strength=0.01, top_k=1,
                    prune_rate=0, t_invert=1.):
        """

        :param out_channels: output channels of the convolutional
            kernel
        :param in_channels: input channels of the convolutional
            kernel
        :param kernel_size: size of the convolutional kernel (int or
            tuple)
        :param stride: stride of the convolutional kernel (int or
            tuple
        :param w_nrm: whether to normalize the weight vectors before
            computing outputs
        :param act: the nonlinear activation function after
            convolution
        :param mode: the learning mode, either 'swta' or 'hpca'
```

```
        :param k: softmax inverse temperature parameter used for
            swta-type learning
        :param patchwise: whether updates for each convolutional
            patch should be computed separately,
        and then aggregated
        :param contrast: coefficient that rescales negative compared
             to positive updates in contrastive-type learning
        :param uniformity: whether to use uniformity weighting in
            contrastive-type learning.
        :param alpha: weighting coefficient between hebbian and
            backprop updates (0 means fully backprop, 1 means fully
            hebbian).
        """
        super(HebbianConv2d, self).__init__()
        self.mode = mode
        self.out_channels = out_channels
        self.in_channels = in_channels
        self.kernel = kernel_size
        self.kernel_size = _pair(kernel_size)
        self.stride = _pair(stride)
        self.dilation = _pair(dilation)
        self.padding = padding
        self.padding_mode = 'reflect'
        if mode == "hard":
            self.padding_mode = 'symmetric'

        self.F_padding = (padding, padding, padding, padding)
        self.groups = 1  # in_channels for depthwise

        weight_range = 25 / math.sqrt(in_channels * kernel_size *
            kernel_size)
        self.weight = nn.Parameter(
            weight_range * torch.randn((out_channels, in_channels //
                self.groups, *self.kernel_size)))

        # Different initialisation schemes for future work

        # self.weight = nn.Parameter(torch.empty(out_channels,
            in_channels // self.groups, *self.kernel_size))
        # init.kaiming_uniform_(self.weight)
        # self.weight = center_surround_init(out_channels,
            in_channels, kernel_size, 1)

        print(self.weight.shape)
        self.w_nrm = w_nrm
        self.act = act

        # BCM parameters
        if mode == "bcm":
            self.theta_decay = 0.5
            self.theta = nn.Parameter(torch.ones(out_channels),
                requires_grad=False)

        self.register_buffer('delta_w', torch.zeros_like(self.weight
            ))
        self.top_k = top_k
        self.patchwise = patchwise
        self.contrast = contrast
```

```
        self.uniformity = uniformity
        self.alpha = alpha
        self.lebesgue_p = 2
        self.prune_rate = prune_rate  # 99% of connections are pruned
        self.t_invert = torch.tensor(t_invert)

        #For presynaptic competition
        self.presynaptic_competition_type = "softmax"
        self.presynaptic_weights = False  # presynaptic competition in forward pass

        # For temporal and statistical thresholds
        self.activation_history = None
        self.temporal_window = 500
        self.competition_k = 2
        self.competition_type = "hard"

        # For surround lateral inhibition kernel
        if self.kernel != 1:
            self.sm_kernel = create_sm_kernel()
            self.register_buffer('surround_kernel', self.sm_kernel)
            # Visualise what the kernel will look like
            self.visualize_surround_modulation_kernel()

        # For direct homeostasis effect
        self.target_activity = 0.1
        self.scaling_rate = 0.001
        self.register_buffer('average_activity', torch.zeros(out_channels))

        self.growth_probability = 0.1
        self.new_synapse_strength = 1
        self.prune_threshold_percentile = 10  # Prune bottom 10% of weights

    def visualize_surround_modulation_kernel(self):
        """
        Visualizes the surround modulation kernel using matplotlib.
        """
        sm_kernel = self.sm_kernel.squeeze().cpu().detach().numpy()
        plt.figure(figsize=(5, 5))
        plt.imshow(sm_kernel, cmap='jet')
        plt.colorbar()
        plt.title('Surround Modulation Kernel')
        plt.show()

    def apply_lebesgue_norm(self, w):
        """Apply Lebesgue norm to weights."""
        return torch.sign(w) * torch.abs(w) ** (self.lebesgue_p - 1)

    def apply_weights(self, x, w):
        """
        Apply convolutional operation with weights to input.
        """
        return F.conv2d(x, w, None, self.stride, 0, self.dilation, groups=self.groups)

```

```
    def update_average_activity(self, y):
        """Update the average activity of neurons."""
        current_activity = y.mean(dim=(0, 2, 3))
        self.average_activity = 0.9 * self.average_activity + 0.1 *
            current_activity

    def synaptic_scaling(self):
        """Apply synaptic scaling to maintain target activity."""
        scale_factor = self.target_activity / (self.average_activity
            + 1e-6)
        self.weight.data *= (1 + self.scaling_rate * (scale_factor -
            1)).view(-1, 1, 1, 1)

    def structural_plasticity(self):
        """Prune sparse weights and create new ones."""
        with torch.no_grad():
            # Pruning step
            prune_threshold = torch.quantile(torch.abs(self.weight),
                self.prune_threshold_percentile / 100)
            weak_synapses = torch.abs(self.weight) < prune_threshold
            self.weight.data[weak_synapses] = 0
            # Growth step
            zero_weights = self.weight.data == 0
            new_synapses = torch.rand_like(self.weight) < self.
                growth_probability
            new_synapses &= zero_weights
            self.weight.data[new_synapses] = torch.randn_like(self.
                weight)[new_synapses] * self.new_synapse_strength

    def cosine(self, x, w):
        """Compute cosine similarity between input and weights."""
        w_normalized = F.normalize(w, p=2, dim=1)
        # conv_output = symmetric_pad(x, self.padding)
        conv_output = F.conv2d(x, w_normalized, None, self.stride,
            0, self.dilation, groups=self.groups)
        x_squared = x.pow(2)
        x_squared_sum = F.conv2d(x_squared, torch.ones_like(w), None
            , self.stride, 0, self.dilation,
                                 self.groups)
        x_norm = torch.sqrt(x_squared_sum + 1e-8)
        cosine_sim = conv_output / x_norm
        return cosine_sim

    def apply_surround_modulation(self, y):
        """Apply surround modulation to the output."""
        return F.conv2d(y, self.sm_kernel.repeat(self.out_channels,
            1, 1, 1),
                        padding=self.sm_kernel.size(-1) // 2, groups
                            =self.out_channels)

    def compute_activation(self, x):
        """Compute the activation of the layer."""
        x = symmetric_pad(x, self.padding)
        w = self.weight
        if self.w_nrm: w = normalize(w, dim=(1, 2, 3))
        if self.presynaptic_weights: w = self.
            compute_presynaptic_competition_global(w)
        # No activation required:
```

```
        # y = self.act(self.apply_weights(x, w))
        # For cosine similarity activation:
        # Must manually change to allow/disable cosine similarity
        y = self.cosine(x, w)
        return x, y, w

    def forward(self, x):
        x,y, w = self.compute_activation(x)
        # Only apply lateral inhibition is spatial dimensions exists
        # Must manually change to allow/disable surround inhibition
        if self.kernel != 1:
            y = self.apply_surround_modulation(y)
        if self.training:
            # Optional mechanisms
            # self.update_average_activity(y)
            # self.synaptic_scaling()
            self.compute_update(x, y, w)
        return y

    def compute_update(self, x, y, weight):
        """Compute weight updates based on the chosen learning mode.
           """
        if self.mode == self.MODE_BASIC_HEBBIAN:
            update = self.update_basic_hebbian(x, y, weight)
        elif self.mode == self.MODE_HARDWT:
            update = self.update_hardwt(x, y, weight)
        elif self.mode == self.MODE_SOFTWTA:
            update = self.update_softwta(x, y, weight)
        elif self.mode == self.MODE_ANTIHARDWT:
            update = self.update_antihardwt(x, y, weight)
        elif self.mode == self.MODE_BCM:
            update = self.update_bcm(x, y, weight)
        elif self.mode == self.MODE_TEMPORAL_COMPETITION:
            update = self.update_temporal_competition(x, y, weight)
        elif self.mode == self.MODE_ADAPTIVE_THRESHOLD:
            update = self.update_adaptive_threshold(x, y, weight)
        else:
            raise NotImplementedError(f"Learning mode {self.mode}
               unavailable for {self.__class__.__name__} layer")
        # Weight Normalization and added to weight change buffer
        update.div_(torch.abs(update).amax() + 1e-30)
        self.delta_w += update

    def update_basic_hebbian(self, x, y, weight):
        """Implement basic Hebbian learning (Grossberg Instar rule).
           """
        yx = self.compute_yx(x, y)
        y_sum = y.sum(dim=(0, 2, 3)).view(-1, 1, 1, 1)
        yw = y_sum * weight
        update = yx - yw
        return update

    def update_hardwt(self, x, y, weight):
        """Implement hard Winner-Take-All (WTA) Hebbian learning."""
        y_wta = y * self.compute_wta_mask(y)
        yx = self.compute_yx(x, y_wta)
        yu = torch.sum(y_wta, dim=(0, 2, 3))
        update = yx - yu.view(-1, 1, 1, 1) * weight
```

```
        return update

    def update_softwta(self, x, y, weight):
        """Implement soft Winner-Take-All Hebbian learning (SoftHebb).""""
        softwta_activs = self.compute_softwta_activations(y)
        yx = self.compute_yx(x, softwta_activs)
        yu = torch.sum(torch.mul(softwta_activs, y), dim=(0, 2, 3))
        update = yx - yu.view(-1, 1, 1, 1) * weight
        return update

    def update_antihardwt(self, x, y, weight):
        """Implement anti-Hebbian hard Winner-Take-All learning."""
        hardwta_activs = self.compute_antihardwta_activations(y)
        yx = self.compute_yx(x, hardwta_activs)
        yu = torch.sum(torch.mul(hardwta_activs, y), dim=(0, 2, 3))
        update = yx - yu.view(-1, 1, 1, 1) * weight
        return update

    def update_bcm(self, x, y, weight):
        """Implement BCM (Bienenstock-Cooper-Munro) learning rule."""
        y_wta = y * self.compute_wta_mask(y)
        y_squared = y_wta.pow(2).mean(dim=(0, 2, 3))
        self.theta.data = (1 - self.theta_decay) * self.theta + self.theta_decay * y_squared
        y_minus_theta = y_wta - self.theta.view(1, -1, 1, 1)
        bcm_factor = y_wta * y_minus_theta
        yx = self.compute_yx(x, bcm_factor)
        update = yx.view(weight.shape)
        return update

    def update_temporal_competition(self, x, y, weight):
        """Implement temporal competition-based learning."""
        self.update_activation_history(y)
        temporal_winners = self.compute_temporal_winners(y)
        y_winners = temporal_winners * y
        y_winners = y_winners * self.apply_competition(y_winners)
        yx = self.compute_yx(x, y_winners)
        y_sum = y_winners.sum(dim=(0, 2, 3)).view(-1, 1, 1, 1)
        update = yx - y_sum * weight
        return update

    def update_adaptive_threshold(self, x, y, weight):
        """Implement adaptive/statistics threshold-based learning."""
        batch_size, out_channels, height_out, width_out = y.shape
        similarities = F.conv2d(x, weight, stride=self.stride, padding=self.padding, groups=self.groups)
        similarities = similarities / (torch.norm(weight.view(out_channels, -1), dim=1).view(1, -1, 1, 1) + 1e-10)
        threshold = self.compute_adaptive_threshold(similarities)
        winners = (similarities > threshold).float()
        y_winners = winners * similarities
        y_winners = y_winners * self.apply_competition(y_winners)
        yx = self.compute_yx(x, y_winners)
        y_sum = y_winners.sum(dim=(0, 2, 3)).view(-1, 1, 1, 1)
        update = yx - y_sum * weight
```

```
336         return update
337
338     def update_activation_history(self, y):
339         """Update the activation history for temporal competition.
                """
340         if self.activation_history is None:
341             self.activation_history = y.detach().clone()
342         else:
343             self.activation_history = torch.cat([self.
                    activation_history, y.detach()], dim=0)
344             if self.activation_history.size(0)>self.temporal_window:
345                 self.activation_history = self.activation_history[-
                        self.temporal_window:]
346
347     def compute_temporal_winners(self, y):
348         """Select winners of temporal competition."""
349         batch_size, out_channels, height_out, width_out = y.shape
350         history_spatial = self.activation_history.view(-1,
                out_channels, height_out, width_out)
351         median_activations = torch.median(history_spatial, dim=0)[0]
352         # Compute threshold for each spatial location
353         temporal_threshold = torch.mean(median_activations, dim=0,
                keepdim=True)
354         return (y > temporal_threshold).float()
355
356     def compute_adaptive_threshold(self, similarities):
357         """Create threshold for statistical thresholding."""
358         mean_sim = similarities.mean(dim=1, keepdim=True)
359         std_sim = similarities.std(dim=1, keepdim=True)
360         return mean_sim + self.competition_k * std_sim
361
362     def apply_competition(self, y):
363         # Hard-WTA or Soft-WTA competition mask for temporal/
                statistical thresholding
364         batch_size, out_channels, height, width = y.shape
365         if self.mode in [self.MODE_TEMPORAL_COMPETITION, self.
                MODE_ADAPTIVE_THRESHOLD]:
366             if self.competition_type == 'hard':
367                 y = y.view(batch_size, out_channels, -1)
368                 top_k_values, top_k_indices = torch.topk(y, self.
                        top_k, dim=1, largest=True, sorted=False)
369                 y_compete = torch.zeros_like(y)
370                 y_compete.scatter_(1, top_k_indices, top_k_values)
371                 return y_compete.view(batch_size, out_channels,
                        height, width)
372             elif self.competition_type == 'soft':
373                 y_flat = y.view(batch_size, out_channels, -1)
374                 y_soft = torch.softmax(self.t_invert * y_flat, dim
                        =1)
375                 return y_soft.view(batch_size, out_channels, height,
                         width)
376         return y
377
378     def compute_yx(self, x, y):
379         """Compute yx term from Grossberg Instar rule"""
380         yx = F.conv2d(x.transpose(0, 1), y.transpose(0, 1), padding
                =0,
```

```
                    stride=self.dilation, dilation=self.stride).
                        transpose(0, 1)
        if self.groups != 1:
            yx = yx.mean(dim=1, keepdim=True)
        return yx

    def compute_wta_mask(self, y):
        """Compute Hard-WTA mask."""
        batch_size, out_channels, height_out, width_out = y.shape
        y_flat = y.transpose(0, 1).reshape(out_channels, -1)
        win_neurons = torch.argmax(y_flat, dim=0)
        wta_mask = F.one_hot(win_neurons, num_classes=out_channels).
            float()
        return wta_mask.transpose(0, 1).view(out_channels,
            batch_size, height_out, width_out).transpose(0, 1)

    def compute_softwta_activations(self, y):
        """Computes SoftHebb mask and Hebb/AntiHebb implementation.
            """
        batch_size, out_channels, height_out, width_out = y.shape
        flat_weighted_inputs = y.transpose(0, 1).reshape(
            out_channels, -1)
        flat_softwta_activs = torch.softmax(self.t_invert *
            flat_weighted_inputs, dim=0)
        flat_softwta_activs = -flat_softwta_activs
        win_neurons = torch.argmax(flat_weighted_inputs, dim=0)
        competing_idx = torch.arange(flat_weighted_inputs.size(1))
        flat_softwta_activs[win_neurons, competing_idx] = -
            flat_softwta_activs[win_neurons, competing_idx]
        return flat_softwta_activs.view(out_channels, batch_size,
            height_out, width_out).transpose(0, 1)

    def compute_antihardwta_activations(self, y):
        # Computes Hard-WTA mask and Hebb/AntiHebb implementation.
        batch_size, out_channels, height_out, width_out = y.shape
        flat_weighted_inputs = y.transpose(0, 1).reshape(
            out_channels, -1)
        win_neurons = torch.argmax(flat_weighted_inputs, dim=0)
        competing_idx = torch.arange(flat_weighted_inputs.size(1))
        anti_hebbian_mask = torch.ones_like(flat_weighted_inputs) *
            -1
        anti_hebbian_mask[win_neurons, competing_idx] = 1
        flat_hardwta_activs = flat_weighted_inputs *
            anti_hebbian_mask
        return flat_hardwta_activs.view(out_channels, batch_size,
            height_out, width_out).transpose(0, 1)

    def compute_presynaptic_competition(self, m):
        # This presynaptic competition promotes diversity among
            output channels.
        m = 1 / (torch.abs(m) + 1e-6)
        if self.presynaptic_competition_type == 'linear':
            return m / (m.sum(dim=0, keepdim=True) + 1e-6)
        elif self.presynaptic_competition_type == 'softmax':
            return F.softmax(m, dim=0)
        elif self.presynaptic_competition_type == 'lp_norm':
            return F.normalize(m, p=2, dim=0)
        else:
```

```
426             raise ValueError(f"Unknown competition type: {self.
                    competition_type}")
427
428     def compute_presynaptic_competition_spatial(self, m):
429         # The presynaptic spatial competition encourages each input-
                output channel pair.
430         m = 1 / (torch.abs(m) + 1e-6)
431         if self.presynaptic_competition_type == 'linear':
432             # Sum across spatial dimensions (last two dimensions)
433             return m / (m.sum(dim=(-2, -1), keepdim=True) + 1e-6)
434         elif self.presynaptic_competition_type == 'softmax':
435             # Reshape to combine spatial dimensions
436             shape = m.shape
437             m_flat = m.view(*shape[:-2], -1)
438             # Apply softmax across spatial dimensions
439             m_comp = F.softmax(m_flat, dim=-1)
440             # Reshape back to original shape
441             return m_comp.view(*shape)
442         elif self.presynaptic_competition_type == 'lp_norm':
443             # Normalize across spatial dimensions
444             return F.normalize(m, p=2, dim=(-2, -1))
445         else:
446             raise ValueError(f"Unknown competition type: {self.
                    presynaptic_competition_type}")
447
448     def compute_presynaptic_competition_input_channels(self, m):
449         # The presynaptic input competition promotes specialisation
                of each output channel across input features.
450         m = 1 / (torch.abs(m) + 1e-6)
451         if self.presynaptic_competition_type == 'linear':
452             # Sum across input channel dimension
453             return m / (m.sum(dim=1, keepdim=True) + 1e-6)
454         elif self.presynaptic_competition_type == 'softmax':
455             # Apply softmax across input channels
456             return F.softmax(m, dim=1)
457         elif self.presynaptic_competition_type == 'lp_norm':
458             # Normalize across input channels
459             return F.normalize(m, p=2, dim=1)
460         else:
461             raise ValueError(f"Unknown competition type: {self.
                    presynaptic_competition_type}")
462
463     def compute_presynaptic_competition_global(self, m):
464         # The global competition creates a more intense competition
                where every weight competes with all others,
465         # potentially leading to very sparse but highly specialised
                connections.
466         m = 1 / (torch.abs(m) + 1e-6)
467         if self.presynaptic_competition_type == 'linear':
468             # Global sum across all dimensions
469             return m / (m.sum() + 1e-6)
470         elif self.presynaptic_competition_type == 'softmax':
471             # Flatten and apply softmax globally
472             m_flat = m.view(-1)
473             return F.softmax(m_flat, dim=0).view(m.shape)
474         elif self.presynaptic_competition_type == 'lp_norm':
475             # Global normalization
476             return F.normalize(m.view(-1), p=2).view(m.shape)
```

```
477             else:
478                 raise ValueError(f"Unknown competition type: {self.
                        presynaptic_competition_type}")
479 
480     @torch.no_grad()
481     def local_update(self):
482         """
483         This function transfers a previously computed weight update,
                stored in buffer self.delta_w, to the gradient
484         self.weight.grad of the weight parameter.
485 
486         This function should be called before optimizer.step(), so
                that the optimizer will use the locally computed
487         update as optimization direction. Local updates can also be
                combined with end-to-end updates by calling this
488         function between loss.backward() and optimizer.step(). loss.
                backward will store the end-to-end gradient in
489         self.weight.grad, and this function combines this value with
                self.delta_w as
490         self.weight.grad = (1 - alpha) * self.weight.grad - alpha *
                self.delta_w
491         Parameter alpha determines the scale of the local update
                compared to the end-to-end gradient in the combination.
492         """
493         if self.weight.grad is None:
494             self.weight.grad = -self.alpha * self.delta_w
495         else:
496             self.weight.grad = (1 - self.alpha) * self.weight.grad -
                    self.alpha * self.delta_w
497         self.delta_w.zero_()
498 
499     # Update method to modify weights without requiring an optimiser
500     # @torch.no_grad()
501     # # Weight Update
502     # def local_update(self):
503     #     new_weight = self.weight + 0.1 * self.alpha * self.delta_w
504     #     # Update weights
505     #     self.weight.copy_(new_weight)
506     #     # self.structural_plasticity()
507     #     self.delta_w.zero_()
```

Listing 2: Calculation of post-synaptic activity in a CNN

A.2.3 Hebb_depthwise.py

```
1 import math
2 
3 import torch
4 import torch.nn as nn
5 import torch.nn.functional as F
6 from torch.nn.modules.utils import _pair
7 import torch.nn.init as init
8 
9 
10 """
11 Uses almost identical code to hebb.py. Please refer to hebb.py for
       additional explanations on code functionality
```

```
Only changes are the competition modes, which apply competition
    across spatial neurons in a filter for channel independence
"""

device = torch.device("cuda" if torch.cuda.is_available() else "cpu"
    )
torch.manual_seed(36)


def normalize(x, dim=None):
    nrm = (x ** 2).sum(dim=dim, keepdim=True) ** 0.5
    nrm[nrm == 0] = 1.
    return x / nrm


def symmetric_pad(x, padding):
    if padding == 0:
        return x
    return F.pad(x, (padding,) * 4, mode='reflect')


def center_surround_init(out_channels, in_channels, kernel_size,
    groups=1):
    # Calculate weight range
    weight_range = 25 / math.sqrt(in_channels * kernel_size *
        kernel_size)
    # Calculate sigma based on kernel size (using equation 3 from
        the paper)
    gamma = torch.empty(out_channels).uniform_(0, 0.5)
    sigma = (kernel_size / 4) * torch.sqrt((1 - gamma ** 2) / (-
        torch.log(gamma)))
    # Create meshgrid for x and y coordinates
    x = torch.linspace(-(kernel_size - 1) / 2, (kernel_size - 1) /
        2, kernel_size)
    y = torch.linspace(-(kernel_size - 1) / 2, (kernel_size - 1) /
        2, kernel_size)
    xx, yy = torch.meshgrid(x, y, indexing='ij')
    # Calculate center and surround Gaussians
    center = torch.exp(-(xx ** 2 + yy ** 2) / (2 * (gamma.view(-1,
        1, 1) * sigma.view(-1, 1, 1)) ** 2))
    surround = torch.exp(-(xx ** 2 + yy ** 2) / (2 * sigma.view(-1,
        1, 1) ** 2))
    # Calculate DoG (Difference of Gaussians)
    dog = center - surround
    # Normalize DoG
    ac = torch.sum(torch.clamp(dog, min=0))
    as_ = torch.sum(-torch.clamp(dog, max=0))
    dog = weight_range * 0.5 * dog / (ac + as_)
    # Assign excitatory (positive) or inhibitory (negative) centers
    center_type = torch.cat([torch.ones(out_channels // 2), -torch.
        ones(out_channels - out_channels // 2)])
    center_type = center_type[torch.randperm(out_channels)].view(-1,
         1, 1)
    dog = dog * center_type
    # Repeat for in_channels and reshape to match conv2d weight
        shape
    dog = dog.unsqueeze(1).repeat(1, in_channels // groups, 1, 1)
```

```
56      dog = dog.reshape(out_channels, in_channels // groups,
            kernel_size, kernel_size)
57      return nn.Parameter(dog)
58
59  def create_sm_kernel(kernel_size=5, sigma_e=1.2, sigma_i=1.4):
60      center = kernel_size // 2
61      x, y = torch.meshgrid(torch.arange(kernel_size), torch.arange(
            kernel_size))
62      x = x.float() - center
63      y = y.float() - center
64      gaussian_e = torch.exp(-(x ** 2 + y ** 2) / (2 * sigma_e ** 2))
65      gaussian_i = torch.exp(-(x ** 2 + y ** 2) / (2 * sigma_i ** 2))
66      dog = gaussian_e / (2 * math.pi * sigma_e ** 2) - gaussian_i /
            (2 * math.pi * sigma_i ** 2)
67      sm_kernel = dog / dog[center, center]
68      return sm_kernel.unsqueeze(0).unsqueeze(0).to(device)
69
70  # Doubts:
71  # Visualizing weights, as separated between channels and spatial
72  #
73
74  class HebbianDepthConv2d(nn.Module):
75      """
76      A 2d convolutional layer that learns through Hebbian plasticity
77      """
78
79      MODE_HPCA = 'hpca'
80      MODE_BASIC_HEBBIAN = 'basic'
81      MODE_WTA = 'wta'
82      MODE_SOFTWTA = 'soft'
83      MODE_BCM = 'bcm'
84      MODE_HARDWT = "hard"
85      MODE_PRESYNAPTIC_COMPETITION = "pre"
86      MODE_TEMPORAL_COMPETITION = "temp"
87      MODE_ADAPTIVE_THRESHOLD = "thresh"
88      MODE_ANTIHARDWT = "antihard"
89
90      def __init__(self, in_channels, out_channels, kernel_size,
            stride=1, dilation=1, padding=0, groups = 1,
91                   w_nrm=False, bias=False, act=nn.Identity(),
92                   mode=MODE_SOFTWTA, k=1, patchwise=True,
93                   contrast=1., uniformity=False, alpha=1.,
                         wta_competition='similarity_spatial',
94                   lateral_competition="filter",
95                   lateral_inhibition_strength=0.01, top_k=1,
                         prune_rate=0, t_invert=1.):
96          """
97
98          :param out_channels: output channels of the convolutional
                kernel
99          :param in_channels: input channels of the convolutional
                kernel
100         :param kernel_size: size of the convolutional kernel (int or
                tuple)
101         :param stride: stride of the convolutional kernel (int or
                tuple
102         :param w_nrm: whether to normalize the weight vectors before
                computing outputs
```

```
103         :param act: the nonlinear activation function after
                convolution
104         :param mode: the learning mode, either 'swta' or 'hpca'
105         :param k: softmax inverse temperature parameter used for
                swta-type learning
106         :param patchwise: whether updates for each convolutional
                patch should be computed separately,
107         and then aggregated
108         :param contrast: coefficient that rescales negative compared
                 to positive updates in contrastive-type learning
109         :param uniformity: whether to use uniformity weighting in
                contrastive-type learning.
110         :param alpha: weighting coefficient between hebbian and
                backprop updates (0 means fully backprop, 1 means fully
                hebbian).
111         """
112         super(HebbianDepthConv2d, self).__init__()
113         # spatial competition is the appropriate comp modes
114         self.mode = mode
115         self.out_channels = out_channels
116         self.in_channels = in_channels
117         self.kernel = kernel_size
118         self.kernel_size = _pair(kernel_size)
119         self.stride = _pair(stride)
120         self.dilation = _pair(dilation)
121         self.padding = padding
122         self.padding_mode = 'reflect'
123         if mode == "hard":
124             self.padding_mode = 'symmetric'
125         self.F_padding = (padding, padding, padding, padding)
126         self.groups = in_channels # in_channels for depthwise
127
128         # # Depthwise separable weights
129         weight_range = 25 / math.sqrt(in_channels * kernel_size *
                kernel_size)
130         self.weight = nn.Parameter(weight_range * torch.randn(
                in_channels, 1, *self.kernel_size))
131
132         # self.weight = nn.Parameter(torch.empty(in_channels, 1, *
                self.kernel_size))
133         # init.kaiming_uniform_(self.weight)
134         # self.weight = center_surround_init(in_channels, 1,
                kernel_size, 1)
135
136         self.w_nrm = w_nrm
137         self.act = act
138         self.theta_decay = 0.5
139         if mode == "bcm":
140             self.theta = nn.Parameter(torch.ones(out_channels))
141
142         self.register_buffer('delta_w', torch.zeros_like(self.weight
                ))
143         self.top_k = top_k
144         self.patchwise = patchwise
145         self.contrast = contrast
146         self.uniformity = uniformity
147         self.alpha = alpha
148         self.wta_competition = wta_competition
```

```
149         self.lateral_inhibition_mode = lateral_competition
150         self.lateral_learning_rate = lateral_inhibition_strength  #
               Adjust as needed
151         self.lebesgue_p = 2
152 
153         self.prune_rate = prune_rate  # 99% of connections are
               pruned
154         self.t_invert = torch.tensor(t_invert)
155 
156         self.presynaptic_competition_type = "softmax"
157         self.presynaptic_weights = False  # presynaptic competition
               in forward pass
158 
159         self.activation_history = None
160         self.temporal_window = 500
161         self.competition_k = 2
162         self.competition_type = "hard"
163 
164         if kernel_size !=1:
165             self.sm_kernel = create_sm_kernel()
166             self.register_buffer('surround_kernel', self.sm_kernel)
167 
168     def apply_lebesgue_norm(self, w):
169         return torch.sign(w) * torch.abs(w) ** (self.lebesgue_p - 1)
170 
171     def cosine(self, x, w):
172         w_normalized = F.normalize(w, p=2, dim=1)
173         conv_output = F.conv2d(x, w_normalized, None, self.stride,
               0, self.dilation, groups=self.groups)
174         x_squared = x.pow(2)
175         x_squared_sum = F.conv2d(x_squared, torch.ones_like(w), None
               , self.stride, 0, self.dilation,
176                                  self.groups)
177         x_norm = torch.sqrt(x_squared_sum + 1e-8)
178         cosine_sim = conv_output / x_norm
179         return cosine_sim
180 
181     def apply_weights(self, x, w):
182         """
183         This function provides the logic for combining input x and
               weight w
184         """
185         # w = self.apply_lebesgue_norm(self.weight)
186         return F.conv2d(x, w, None, self.stride, 0, self.dilation,
               groups=self.groups)
187 
188     def apply_surround_modulation(self, y):
189         return F.conv2d(y, self.sm_kernel.repeat(self.out_channels,
               1, 1, 1),
190                         padding=self.sm_kernel.size(-1) // 2, groups
                            =self.out_channels)
191 
192     def compute_activation(self, x):
193         x = symmetric_pad(x, self.padding)
194         w = self.weight
195         if self.w_nrm: w = normalize(w, dim=(1, 2, 3))
196         if self.presynaptic_weights: w = self.
               compute_presynaptic_competition(w)
```

```
197         # y_depthwise = self.act(self.apply_weights(x, w))
198         # For cosine similarity activation if cosine is to be used
                for next layer
199         y_depthwise = self.cosine(x,w)
200         return x, y_depthwise, w
201
202     def forward(self, x):
203         x, y_depthwise, w = self.compute_activation(x)
204         if self.kernel !=1:
205             y_depthwise = self.apply_surround_modulation(y_depthwise
                    )
206         if self.training:
207             self.compute_update(x, y_depthwise, w)
208         return y_depthwise
209
210     def compute_update(self, x, y, weight):
211         if self.mode == self.MODE_BASIC_HEBBIAN:
212             update = self.update_basic_hebbian(x, y, weight)
213         elif self.mode == self.MODE_HARDWT:
214             update = self.update_hardwt(x, y, weight)
215         elif self.mode == self.MODE_SOFTWTA:
216             update = self.update_softwta(x, y, weight)
217         elif self.mode == self.MODE_BCM:
218             update = self.update_bcm(x, y, weight)
219         elif self.mode == self.MODE_TEMPORAL_COMPETITION:
220             update = self.update_temporal_competition(x, y, weight)
221         elif self.mode == self.MODE_ADAPTIVE_THRESHOLD:
222             update = self.update_adaptive_threshold(x, y, weight)
223         else:
224             raise NotImplementedError(f"Learning mode {self.mode}
                    unavailable for {self.__class__.__name__} layer")
225
226         # Weight Normalization and added to weight change buffer
227         update.div_(torch.abs(update).amax() + 1e-30)
228         self.delta_w += update
229
230     def update_basic_hebbian(self, x, y, weight):
231         yx = self.compute_yx(x, y)
232         y_sum = y.sum(dim=(0, 2, 3)).view(self.in_channels, 1, 1, 1)
233         yw = y_sum * weight
234         return yx - yw
235
236     def update_hardwt(self, x, y, weight):
237         y_wta = y * self.compute_wta_mask(y)
238         yx = self.compute_yx(x, y_wta)
239         yu = torch.sum(y_wta, dim=(0, 2, 3)).view(self.in_channels,
                1, 1, 1)
240         return yx - yu * weight
241
242     def update_softwta(self, x, y, weight):
243         softwta_activs = self.compute_softwta_activations(y)
244         yx = self.compute_yx(x, softwta_activs)
245         yu = torch.sum(torch.mul(softwta_activs, y), dim=(0, 2, 3)).
                view(self.in_channels, 1, 1, 1)
246         return yx - yu * weight
247
248     def update_bcm(self, x, y, weight):
249         batch_size, out_channels, height, width = y.shape
```

```
250         y_wta = y * self.compute_wta_mask(y)
251         # Compute squared activation for each spatial location and
                channel
252         y_squared = y_wta.pow(2)
253         # Reshape theta to match spatial dimensions
254         theta_spatial = self.theta.view(1, -1, 1, 1).expand(1,
                out_channels, height, width)
255         # Update theta for each spatial location
256         theta_update = self.theta_decay * (y_squared - theta_spatial
                )
257         self.theta.data += theta_update.mean(dim=(0, 2, 3))
258         # Compute BCM updates for each spatial location
259         y_minus_theta = y_wta - theta_spatial
260         bcm_factor = y_wta * y_minus_theta
261         yx = self.compute_yx(x, bcm_factor)
262         update = yx.view(weight.shape)
263         return update
264
265     def update_temporal_competition(self, x, y, weight):
266         batch_size, out_channels, height_out, width_out = y.shape
267         self.update_activation_history(y)
268         temporal_winners = self.compute_temporal_winners(y)
269         y_winners = temporal_winners * y
270         y_winners = self.apply_competition(y_winners, batch_size,
                out_channels)
271         yx = self.compute_yx(x, y_winners)
272         y_sum = y_winners.sum(dim=(0, 2, 3)).view(self.in_channels,
                1, 1, 1)
273         update = yx - y_sum * weight
274         return update
275
276     def update_adaptive_threshold(self, x, y, weight):
277         batch_size, out_channels, height_out, width_out = y.shape
278         similarities = F.conv2d(x, weight, stride=self.stride,
                padding=self.padding, groups=self.groups)
279         similarities = similarities / (torch.norm(weight.view(
                out_channels, -1), dim=1).view(1, -1, 1, 1) + 1e-10)
280         threshold = self.compute_adaptive_threshold(similarities)
281         winners = (similarities > threshold).float()
282         y_winners = winners * similarities
283         y_winners = self.apply_competition(y_winners, batch_size,
                out_channels)
284         yx = self.compute_yx(x, y_winners)
285         y_sum = y_winners.sum(dim=(0, 2, 3)).view(self.in_channels,
                1, 1, 1)
286         update = yx - y_sum * weight
287         return update
288
289     def compute_yx(self, x, y):
290         yx = F.conv2d(x.transpose(0, 1), y.transpose(0, 1), padding
                =0,
291                       stride=self.dilation, dilation=self.stride).
                          transpose(0, 1)
292         yx = yx.diagonal(dim1=0, dim2=1).permute(2, 0, 1).unsqueeze
                (1)
293         return yx
294
295     def compute_wta_mask(self, y):
```

```
296         batch_size, out_channels, height_out, width_out = y.shape
297         # WTA competition within each channel
298         y_flat = y.view(batch_size, out_channels, -1)
299         win_neurons = torch.argmax(y_flat, dim=2)
300         wta_mask = F.one_hot(win_neurons, num_classes=height_out *
                width_out).float()
301         return wta_mask.view(batch_size, out_channels, height_out,
                width_out)
302
303     def compute_softwta_activations(self, y):
304         # Competition and anti-Hebbian learning for y_depthwise
305         batch_size, in_channels, height_depthwise, width_depthwise =
                y.shape
306         # Reshape to apply softmax within each channel
307         y_depthwise_reshaped = y.view(batch_size, in_channels, -1)
308         # Apply softmax within each channel
309         flat_softwta_activs_depthwise = torch.softmax(self.t_invert
                * y_depthwise_reshaped, dim=2)
310         # Turn all postsynaptic activations into anti-Hebbian
311         flat_softwta_activs_depthwise = -
                flat_softwta_activs_depthwise
312         # Find winners within each channel
313         win_neurons_depthwise = torch.argmax(y_depthwise_reshaped,
                dim=2)
314         # Create a mask to flip the sign of winning neurons
315         mask = torch.zeros_like(flat_softwta_activs_depthwise)
316         mask.scatter_(2, win_neurons_depthwise.unsqueeze(2), 1)
317         # Flip the sign of winning neurons
318         flat_softwta_activs_depthwise =
                flat_softwta_activs_depthwise * (1 - 2 * mask)
319         # Reshape back to original shape
320         return flat_softwta_activs_depthwise.view(batch_size,
                in_channels, height_depthwise,
321                                                                 width_dept
                                                                    )
322
323     def update_activation_history(self, y):
324         if self.activation_history is None:
325             self.activation_history = y.detach().clone()
326         else:
327             self.activation_history = torch.cat([self.
                    activation_history, y.detach()], dim=0)
328             if self.activation_history.size(0) > self.
                    temporal_window:
329                 self.activation_history = self.activation_history[-
                        self.temporal_window:]
330
331     def compute_temporal_winners(self, y):
332         batch_size, out_channels, height_out, width_out = y.shape
333         history_spatial = self.activation_history.view(-1,
                out_channels, height_out, width_out)
334         median_activations = torch.median(history_spatial, dim=0)[0]
335         temporal_threshold = torch.mean(median_activations, dim=(1,
                2), keepdim=True)
336         return (median_activations > temporal_threshold).float()
337
338     def compute_adaptive_threshold(self, similarities):
```

```
339         mean_sim = similarities.mean(dim=(2, 3), keepdim=True)
340         std_sim = similarities.std(dim=(2, 3), keepdim=True)
341         return mean_sim + self.competition_k * std_sim
342 
343     def apply_competition(self, y, batch_size, out_channels):
344         if self.mode in [self.MODE_TEMPORAL_COMPETITION, self.
                MODE_ADAPTIVE_THRESHOLD]:
345             if self.competition_type == 'hard':
346                 y = y.view(batch_size, out_channels, -1)
347                 top_k_indices = torch.topk(y, self.top_k, dim=2,
                        largest=True, sorted=False).indices
348                 y_compete = torch.zeros_like(y)
349                 y_compete.scatter_(2, top_k_indices, y.gather(2,
                        top_k_indices))
350                 return y_compete.view_as(y)
351             elif self.competition_type == 'soft':
352                 return torch.softmax(self.t_invert * y.view(
                        batch_size, out_channels, -1), dim=2).view_as(y)
353         return y
354 
355     @torch.no_grad()
356     def local_update(self):
357         """
358         This function transfers a previously computed weight update,
                stored in buffer self.delta_w, to the gradient
359         self.weight.grad of the weight parameter.
360 
361         This function should be called before optimizer.step(), so
                that the optimizer will use the locally computed
362         update as optimization direction. Local updates can also be
                combined with end-to-end updates by calling this
363         function between loss.backward() and optimizer.step(). loss.
                backward will store the end-to-end gradient in
364         self.weight.grad, and this function combines this value with
                self.delta_w as
365         self.weight.grad = (1 - alpha) * self.weight.grad - alpha *
                self.delta_w
366         Parameter alpha determines the scale of the local update
                compared to the end-to-end gradient in the combination.
367         """
368         if self.weight.grad is None:
369             self.weight.grad = -self.alpha * self.delta_w
370         else:
371             self.weight.grad = (1 - self.alpha) * self.weight.grad -
                    self.alpha * self.delta_w
372         self.delta_w.zero_()
373 
374     # @torch.no_grad()
375     # # Weight Update
376     # def local_update(self):
377     #     new_weight = self.weight + 0.1 * self.alpha * self.delta_w
378     #     # Update weights
379     #     self.weight.copy_(new_weight)
380     #     # self.structural_plasticity()
381     #     self.delta_w.zero_()
```

A.2.4 Hebb_abs.py

```python
import math

import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.nn.modules.utils import _pair
import matplotlib.pyplot as plt

import torch.nn.init as init

"""
Uses almost identical code to hebb.py. Please refer to hebb.py for additional explanations on code functionality
Only changes are the initialisation and update of weights, to ensure adherence to Dale's Principle
"""

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
torch.manual_seed(0)


def normalize(x, dim=None):
    nrm = (x ** 2).sum(dim=dim, keepdim=True) ** 0.5
    nrm[nrm == 0] = 1.
    return x / nrm


def symmetric_pad(x, padding):
    if padding == 0:
        return x
    return F.pad(x, (padding,) * 4, mode='reflect')


def create_sm_kernel(kernel_size=5, sigma_e=1.2, sigma_i=1.4):
    """
    Create a surround modulation kernel that matches the paper's specifications.
    :param kernel_size: Size of the SM kernel.
    :param sigma_e: Standard deviation for the excitatory Gaussian.
    :param sigma_i: Standard deviation for the inhibitory Gaussian.
    :return: A normalized SM kernel.
    """
    center = kernel_size // 2
    x, y = torch.meshgrid(torch.arange(kernel_size), torch.arange(kernel_size), indexing="ij")
    x = x.float() - center
    y = y.float() - center
    # Compute the excitatory and inhibitory Gaussians
    gaussian_e = torch.exp(-(x ** 2 + y ** 2) / (2 * sigma_e ** 2))
    gaussian_i = torch.exp(-(x ** 2 + y ** 2) / (2 * sigma_i ** 2))
    # Compute the Difference of Gaussians (DoG)
    dog = gaussian_e / (2 * math.pi * sigma_e ** 2) - gaussian_i / (2 * math.pi * sigma_i ** 2)
    # Normalize the DoG so that the center value is 1
    sm_kernel = dog / dog[center, center]
    return sm_kernel.unsqueeze(0).unsqueeze(0).to(device)
```

```
52
53
54 class HebbianConv2d(nn.Module):
55     """
56     A 2d convolutional layer that learns through Hebbian plasticity
57     """
58
59     MODE_HPCA = 'hpca'
60     MODE_BASIC_HEBBIAN = 'basic'
61     MODE_WTA = 'wta'
62     MODE_SOFTWTA = 'soft'
63     MODE_BCM = 'bcm'
64     MODE_HARDWT = "hard"
65     MODE_PRESYNAPTIC_COMPETITION = "pre"
66     MODE_TEMPORAL_COMPETITION = "temp"
67     MODE_ADAPTIVE_THRESHOLD = "thresh"
68     MODE_ANTIHARDWT = "antihard"
69
70     def __init__(self, in_channels, out_channels, kernel_size,
           stride=1, dilation=1, padding=0, groups=1,
71                  w_nrm=False, bias=False, act=nn.Identity(),
72                  mode=MODE_SOFTWTA, k=1, patchwise=True,
73                  contrast=1., uniformity=False, alpha=1.,
                        wta_competition='similarity_spatial',
74                  lateral_competition="combined",
75                  lateral_inhibition_strength=0.01, top_k=1,
                        prune_rate=0, t_invert=1.):
76         """
77
78         :param out_channels: output channels of the convolutional
               kernel
79         :param in_channels: input channels of the convolutional
               kernel
80         :param kernel_size: size of the convolutional kernel (int or
                tuple)
81         :param stride: stride of the convolutional kernel (int or
               tuple
82         :param w_nrm: whether to normalize the weight vectors before
                computing outputs
83         :param act: the nonlinear activation function after
               convolution
84         :param mode: the learning mode, either 'swta' or 'hpca'
85         :param k: softmax inverse temperature parameter used for
               swta-type learning
86         :param patchwise: whether updates for each convolutional
               patch should be computed separately,
87         and then aggregated
88         :param contrast: coefficient that rescales negative compared
                to positive updates in contrastive-type learning
89         :param uniformity: whether to use uniformity weighting in
               contrastive-type learning.
90         :param alpha: weighting coefficient between hebbian and
               backprop updates (0 means fully backprop, 1 means fully
               hebbian).
91         """
92         super(HebbianConv2d, self).__init__()
93         # spatial competition is the appropriate comp modes
94         self.mode = mode
```

```
        self.out_channels = out_channels
        self.in_channels = in_channels
        self.kernel = kernel_size
        self.kernel_size = _pair(kernel_size)
        self.stride = _pair(stride)
        self.dilation = _pair(dilation)
        self.padding = padding
        self.padding_mode = 'reflect'
        if mode == "hard":
            self.padding_mode = 'symmetric'

        self.F_padding = (padding, padding, padding, padding)
        self.groups = 1  # in_channels for depthwise

        weight_range = 25 / math.sqrt(in_channels * kernel_size *
            kernel_size)
        self.weight = nn.Parameter(weight_range * torch.abs(torch.
            randn((out_channels, in_channels // self.groups, *self.
            kernel_size))))

        print(self.weight.shape)
        self.w_nrm = w_nrm
        self.act = act
        self.theta_decay = 0.5
        if mode == "bcm":
            self.theta = nn.Parameter(torch.ones(out_channels),
                requires_grad=False)

        self.register_buffer('delta_w', torch.zeros_like(self.weight
            ))
        self.top_k = top_k
        self.patchwise = patchwise
        self.contrast = contrast
        self.uniformity = uniformity
        self.alpha = alpha
        self.wta_competition = wta_competition
        self.lateral_inhibition_mode = lateral_competition
        self.lateral_learning_rate = lateral_inhibition_strength  #
            Adjust as needed
        self.lebesgue_p = 2

        self.prune_rate = prune_rate  # 99% of connections are
            pruned
        self.t_invert = torch.tensor(t_invert)

        self.presynaptic_competition_type = "softmax"
        self.presynaptic_weights = False  # presynaptic competition
            in forward pass

        self.activation_history = None
        self.temporal_window = 500
        self.competition_k = 2
        self.competition_type = "hard"

        if self.kernel != 1:
            self.sm_kernel = create_sm_kernel()
            self.register_buffer('surround_kernel', self.sm_kernel)
            self.visualize_surround_modulation_kernel()
```

```
145
146         self.target_activity = 0.1
147         self.scaling_rate = 0.001
148         self.register_buffer('average_activity', torch.zeros(
                out_channels))
149
150         self.growth_probability = 0.1
151         self.new_synapse_strength = 1
152         self.prune_threshold_percentile = 10  # Prune bottom 10% of
                weights
153
154     def visualize_surround_modulation_kernel(self):
155         """
156         Visualizes the surround modulation kernel using matplotlib.
157         """
158         sm_kernel = self.sm_kernel.squeeze().cpu().detach().numpy()
159         plt.figure(figsize=(5, 5))
160         plt.imshow(sm_kernel, cmap='jet')
161         plt.colorbar()
162         plt.title('Surround Modulation Kernel')
163         plt.show()
164
165     def apply_lebesgue_norm(self, w):
166         return torch.sign(w) * torch.abs(w) ** (self.lebesgue_p - 1)
167
168     def apply_weights(self, x, w):
169         """
170         This function combines input x and weight w
171         """
172         return F.conv2d(x, w, None, self.stride, 0, self.dilation,
                groups=self.groups)
173
174     def update_average_activity(self, y):
175         current_activity = y.mean(dim=(0, 2, 3))
176         self.average_activity = 0.9 * self.average_activity + 0.1 *
                current_activity
177
178     def synaptic_scaling(self):
179         scale_factor = self.target_activity / (self.average_activity
                + 1e-6)
180         self.weight.data *= (1 + self.scaling_rate * (scale_factor -
                1)).view(-1, 1, 1, 1)
181
182     def structural_plasticity(self):
183         with torch.no_grad():
184             # Pruning step
185             prune_threshold = torch.quantile(torch.abs(self.weight),
                    self.prune_threshold_percentile / 100)
186             weak_synapses = torch.abs(self.weight) < prune_threshold
187             self.weight.data[weak_synapses] = 0
188             # Growth step
189             zero_weights = self.weight.data == 0
190             new_synapses = torch.rand_like(self.weight) < self.
                    growth_probability
191             new_synapses &= zero_weights
192             self.weight.data[new_synapses] = torch.randn_like(self.
                    weight)[new_synapses] * self.new_synapse_strength
193
```

```
194     def cosine(self, x, w):
195         w_normalized = F.normalize(w, p=2, dim=1)
196         conv_output = F.conv2d(x, w_normalized, None, self.stride,
                0, self.dilation, groups=self.groups)
197         x_squared = x.pow(2)
198         x_squared_sum = F.conv2d(x_squared, torch.ones_like(w), None
                , self.stride, 0, self.dilation,
199                                  self.groups)
200         x_norm = torch.sqrt(x_squared_sum + 1e-8)
201         cosine_sim = conv_output / x_norm
202         return cosine_sim
203 
204     def apply_surround_modulation(self, y):
205         return F.conv2d(y, self.sm_kernel.repeat(self.out_channels,
                1, 1, 1),
206                         padding=self.sm_kernel.size(-1) // 2, groups
                            =self.out_channels)
207 
208     def compute_activation(self, x):
209         x = symmetric_pad(x, self.padding)
210         w = self.weight.abs()
211         if self.w_nrm: w = normalize(w, dim=(1, 2, 3))
212         if self.presynaptic_weights: w = self.
                compute_presynaptic_competition_global(w)
213         # y = self.act(self.apply_weights(x, w))
214         # For cosine similarity activation if cosine is to be used
                for next layer
215         y = self.cosine(x, w)
216         return x,y, w
217 
218     def forward(self, x):
219         x,y, w = self.compute_activation(x)
220         if self.kernel != 1:
221             y = self.apply_surround_modulation(y)
222         if self.training:
223             # self.update_average_activity(y)
224             # self.synaptic_scaling()
225             self.compute_update(x, y, w)
226         return y
227 
228     def compute_update(self, x, y, weight):
229         if self.mode == self.MODE_BASIC_HEBBIAN:
230             update = self.update_basic_hebbian(x, y, weight)
231         elif self.mode == self.MODE_HARDWT:
232             update = self.update_hardwt(x, y, weight)
233         elif self.mode == self.MODE_SOFTWTA:
234             update = self.update_softwta(x, y, weight)
235         elif self.mode == self.MODE_BCM:
236             update = self.update_bcm(x, y, weight)
237         else:
238             raise NotImplementedError(f"Learning mode {self.mode}
                    unavailable for {self.__class__.__name__} layer")
239         # Weight Normalization and added to weight change buffer
240         update.div_(torch.abs(update).amax() + 1e-30)
241         self.delta_w += update
242 
243     def update_basic_hebbian(self, x, y, weight):
244         # Grossberg Instar rule
```

```
245         yx = self.compute_yx(x, y)
246         y_sum = y.sum(dim=(0, 2, 3)).view(-1, 1, 1, 1)
247         yw = y_sum * weight
248         update = yx - yw
249         return update
250 
251     def update_hardwt(self, x, y, weight):
252         # Grossberg Instar with wta mask
253         y_wta = y * self.compute_wta_mask(y)
254         yx = self.compute_yx(x, y_wta)
255         yu = torch.sum(y_wta, dim=(0, 2, 3))
256         update = yx - yu.view(-1, 1, 1, 1) * weight
257         return update
258 
259     def update_softwta(self, x, y, weight):
260         # SoftHebb Grossberg Instar Variation
261         softwta_activs = self.compute_softwta_activations(y)
262         yx = self.compute_yx(x, softwta_activs)
263         yu = torch.sum(torch.mul(softwta_activs, y), dim=(0, 2, 3))
264         update = yx - yu.view(-1, 1, 1, 1) * weight
265         return update
266 
267     def update_bcm(self, x, y, weight):
268         # BCM learning rule with WTA mask (remove if not needed)
269         y_wta = y * self.compute_wta_mask(y)
270         y_squared = y_wta.pow(2).mean(dim=(0, 2, 3))
271         self.theta.data = (1 - self.theta_decay) * self.theta + self
                .theta_decay * y_squared
272         y_minus_theta = y_wta - self.theta.view(1, -1, 1, 1)
273         bcm_factor = y_wta * y_minus_theta
274         yx = self.compute_yx(x, bcm_factor)
275         update = yx.view(weight.shape)
276         return update
277 
278     def compute_yx(self, x, y):
279         # Computes common y*w term from Grossberg Instar
280         yx = F.conv2d(x.transpose(0, 1), y.transpose(0, 1), padding
                =0,
281                       stride=self.dilation, dilation=self.stride).
                          transpose(0, 1)
282         if self.groups != 1:
283             yx = yx.mean(dim=1, keepdim=True)
284         return yx
285 
286     def compute_wta_mask(self, y):
287         # Computes WTA Mask
288         batch_size, out_channels, height_out, width_out = y.shape
289         y_flat = y.transpose(0, 1).reshape(out_channels, -1)
290         win_neurons = torch.argmax(y_flat, dim=0)
291         wta_mask = F.one_hot(win_neurons, num_classes=out_channels).
                float()
292         return wta_mask.transpose(0, 1).view(out_channels,
                batch_size, height_out, width_out).transpose(0, 1)
293 
294     def compute_softwta_activations(self, y):
295         # Computes SoftHebb mask and Hebb/AntiHebb implementation
296         batch_size, out_channels, height_out, width_out = y.shape
```

```
297         flat_weighted_inputs = y.transpose(0, 1).reshape(
                out_channels, -1)
298         flat_softwta_activs = torch.softmax(self.t_invert *
                flat_weighted_inputs, dim=0)
299         flat_softwta_activs = -flat_softwta_activs
300         win_neurons = torch.argmax(flat_weighted_inputs, dim=0)
301         competing_idx = torch.arange(flat_weighted_inputs.size(1))
302         flat_softwta_activs[win_neurons, competing_idx] = -
                flat_softwta_activs[win_neurons, competing_idx]
303         return flat_softwta_activs.view(out_channels, batch_size,
                height_out, width_out).transpose(0, 1)
304
305     def compute_antihardwta_activations(self, y):
306         # Computes Hard-WTA mask and Hebb/AntiHebb implementation
307         batch_size, out_channels, height_out, width_out = y.shape
308         flat_weighted_inputs = y.transpose(0, 1).reshape(
                out_channels, -1)
309         win_neurons = torch.argmax(flat_weighted_inputs, dim=0)
310         competing_idx = torch.arange(flat_weighted_inputs.size(1))
311         anti_hebbian_mask = torch.ones_like(flat_weighted_inputs) *
                -1
312         anti_hebbian_mask[win_neurons, competing_idx] = 1
313         flat_hardwta_activs = flat_weighted_inputs *
                anti_hebbian_mask
314         return flat_hardwta_activs.view(out_channels, batch_size,
                height_out, width_out).transpose(0, 1)
315
316     # Weight update which ensures only excitatory weights
317     @torch.no_grad()
318     # Weight Update
319     def local_update(self):
320         new_weight = self.weight + 0.1 * self.alpha * self.delta_w
321         # Update weights
322         self.weight.copy_(new_weight.abs())
323         self.delta_w.zero_()
```

A.2.5 Hebb_abs_depthwise.py

```
1  import math
2
3  import torch
4  import torch.nn as nn
5  import torch.nn.functional as F
6  from torch.nn.modules.utils import _pair
7  import torch.nn.init as init
8
9
10 """
11 Uses almost identical code to hebb.py. Please refer to hebb.py for
       additional explanations on code functionality
12 Only changes are the competition modes, which apply competition
       across spatial neurons in a filter for channel independence
13 Cahnegs also in initialisation and update, similar to hebb_abs.py
14 """
15
16 device = torch.device("cuda" if torch.cuda.is_available() else "cpu"
       )
17 torch.manual_seed(0)
```

```
18
19
20 def normalize(x, dim=None):
21     nrm = (x ** 2).sum(dim=dim, keepdim=True) ** 0.5
22     nrm[nrm == 0] = 1.
23     return x / nrm
24
25
26 def symmetric_pad(x, padding):
27     if padding == 0:
28         return x
29     return F.pad(x, (padding,) * 4, mode='reflect')
30
31
32 def center_surround_init(out_channels, in_channels, kernel_size,
       groups=1):
33     # Calculate weight range
34     weight_range = 25 / math.sqrt(in_channels * kernel_size *
           kernel_size)
35
36     # Calculate sigma based on kernel size (using equation 3 from
           the paper)
37     gamma = torch.empty(out_channels).uniform_(0, 0.5)
38     sigma = (kernel_size / 4) * torch.sqrt((1 - gamma ** 2) / (-
           torch.log(gamma)))
39
40     # Create meshgrid for x and y coordinates
41     x = torch.linspace(-(kernel_size - 1) / 2, (kernel_size - 1) /
           2, kernel_size)
42     y = torch.linspace(-(kernel_size - 1) / 2, (kernel_size - 1) /
           2, kernel_size)
43     xx, yy = torch.meshgrid(x, y, indexing='ij')
44
45     # Calculate center and surround Gaussians
46     center = torch.exp(-(xx ** 2 + yy ** 2) / (2 * (gamma.view(-1,
           1, 1) * sigma.view(-1, 1, 1)) ** 2))
47     surround = torch.exp(-(xx ** 2 + yy ** 2) / (2 * sigma.view(-1,
           1, 1) ** 2))
48
49     # Calculate DoG (Difference of Gaussians)
50     dog = center - surround
51
52     # Normalize DoG
53     ac = torch.sum(torch.clamp(dog, min=0))
54     as_ = torch.sum(-torch.clamp(dog, max=0))
55     dog = weight_range * 0.5 * dog / (ac + as_)
56
57     # Assign excitatory (positive) or inhibitory (negative) centers
58     center_type = torch.cat([torch.ones(out_channels // 2), -torch.
           ones(out_channels - out_channels // 2)])
59     center_type = center_type[torch.randperm(out_channels)].view(-1,
            1, 1)
60     dog = dog * center_type
61
62     # Repeat for in_channels and reshape to match conv2d weight
           shape
63     dog = dog.unsqueeze(1).repeat(1, in_channels // groups, 1, 1)
```

```
64      dog = dog.reshape(out_channels, in_channels // groups,
            kernel_size, kernel_size)
65
66      return nn.Parameter(dog)
67
68  def create_sm_kernel(kernel_size=5, sigma_e=1.2, sigma_i=1.4):
69      center = kernel_size // 2
70      x, y = torch.meshgrid(torch.arange(kernel_size), torch.arange(
            kernel_size))
71      x = x.float() - center
72      y = y.float() - center
73
74      gaussian_e = torch.exp(-(x ** 2 + y ** 2) / (2 * sigma_e ** 2))
75      gaussian_i = torch.exp(-(x ** 2 + y ** 2) / (2 * sigma_i ** 2))
76
77      dog = gaussian_e / (2 * math.pi * sigma_e ** 2) - gaussian_i /
            (2 * math.pi * sigma_i ** 2)
78      sm_kernel = dog / dog[center, center]
79
80      return sm_kernel.unsqueeze(0).unsqueeze(0).to(device)
81
82
83  class HebbianDepthConv2d(nn.Module):
84      """
85      A 2d convolutional layer that learns through Hebbian plasticity
86      """
87
88      MODE_HPCA = 'hpca'
89      MODE_BASIC_HEBBIAN = 'basic'
90      MODE_WTA = 'wta'
91      MODE_SOFTWTA = 'soft'
92      MODE_BCM = 'bcm'
93      MODE_HARDWT = "hard"
94      MODE_PRESYNAPTIC_COMPETITION = "pre"
95      MODE_TEMPORAL_COMPETITION = "temp"
96      MODE_ADAPTIVE_THRESHOLD = "thresh"
97      MODE_ANTIHARDWT = "antihard"
98
99      def __init__(self, in_channels, out_channels, kernel_size,
            stride=1, dilation=1, padding=0, groups = 1,
100                 w_nrm=False, bias=False, act=nn.Identity(),
101                 mode=MODE_SOFTWTA, k=1, patchwise=True,
102                 contrast=1., uniformity=False, alpha=1.,
                        wta_competition='similarity_spatial',
103                 lateral_competition="filter",
104                 lateral_inhibition_strength=0.01, top_k=1,
                        prune_rate=0, t_invert=1.):
105          """
106
107          :param out_channels: output channels of the convolutional
                kernel
108          :param in_channels: input channels of the convolutional
                kernel
109          :param kernel_size: size of the convolutional kernel (int or
                 tuple)
110          :param stride: stride of the convolutional kernel (int or
                tuple
```

```
111         :param w_nrm: whether to normalize the weight vectors before
                computing outputs
112         :param act: the nonlinear activation function after
               convolution
113         :param mode: the learning mode, either 'swta' or 'hpca'
114         :param k: softmax inverse temperature parameter used for
               swta-type learning
115         :param patchwise: whether updates for each convolutional
               patch should be computed separately,
116         and then aggregated
117         :param contrast: coefficient that rescales negative compared
                to positive updates in contrastive-type learning
118         :param uniformity: whether to use uniformity weighting in
               contrastive-type learning.
119         :param alpha: weighting coefficient between hebbian and
               backprop updates (0 means fully backprop, 1 means fully
               hebbian).
120         """
121         super(HebbianDepthConv2d, self).__init__()
122         # spatial competition is the appropriate comp modes
123         self.mode = mode
124         self.out_channels = out_channels
125         self.in_channels = in_channels
126         self.kernel = kernel_size
127         self.kernel_size = _pair(kernel_size)
128         self.stride = _pair(stride)
129         self.dilation = _pair(dilation)
130         self.padding = padding
131         self.padding_mode = 'reflect'
132         if mode == "hard":
133             self.padding_mode = 'symmetric'
134         self.F_padding = (padding, padding, padding, padding)
135         self.groups = in_channels # in_channels for depthwise
136
137         # Depthwise separable weights
138         weight_range = 25 / math.sqrt(in_channels * kernel_size *
               kernel_size)
139         self.weight = nn.Parameter(weight_range * torch.abs(torch.
               randn(in_channels, 1, *self.kernel_size)))
140
141         # self.weight = center_surround_init(in_channels, 1,
               kernel_size, 1)
142
143         self.w_nrm = w_nrm
144         self.act = act
145         self.theta_decay = 0.5
146         if mode == "bcm":
147             self.theta = nn.Parameter(torch.ones(out_channels))
148
149         self.register_buffer('delta_w', torch.zeros_like(self.weight
               ))
150         self.top_k = top_k
151         self.patchwise = patchwise
152         self.contrast = contrast
153         self.uniformity = uniformity
154         self.alpha = alpha
155         self.wta_competition = wta_competition
156         self.lateral_inhibition_mode = lateral_competition
```

```
157         self.lateral_learning_rate = lateral_inhibition_strength  #
               Adjust as needed
158         self.lebesgue_p = 2
159
160         self.prune_rate = prune_rate  # 99% of connections are
               pruned
161         self.t_invert = torch.tensor(t_invert)
162
163         self.presynaptic_competition_type = "softmax"
164         self.presynaptic_weights = False  # presynaptic competition
               in forward pass
165
166         self.activation_history = None
167         self.temporal_window = 500
168         self.competition_k = 2
169         self.competition_type = "hard"
170
171         if kernel_size !=1:
172             self.sm_kernel = create_sm_kernel()
173             self.register_buffer('surround_kernel', self.sm_kernel)
174
175     def apply_lebesgue_norm(self, w):
176         return torch.sign(w) * torch.abs(w) ** (self.lebesgue_p - 1)
177
178     def cosine(self, x, w):
179         w_normalized = F.normalize(w, p=2, dim=1)
180         conv_output = F.conv2d(x, w_normalized, None, self.stride,
               0, self.dilation, groups=self.groups)
181         x_squared = x.pow(2)
182         x_squared_sum = F.conv2d(x_squared, torch.ones_like(w), None
               , self.stride, 0, self.dilation,
183                                  self.groups)
184         x_norm = torch.sqrt(x_squared_sum + 1e-8)
185         cosine_sim = conv_output / x_norm
186         return cosine_sim
187
188     def apply_weights(self, x, w):
189         """
190         This function combines input x and weight w
191         """
192         # w = self.apply_lebesgue_norm(self.weight)
193
194         return F.conv2d(x, w, None, self.stride, 0, self.dilation,
               groups=self.groups)
195
196     def apply_surround_modulation(self, y):
197         return F.conv2d(y, self.sm_kernel.repeat(self.out_channels,
               1, 1, 1),
198                         padding=self.sm_kernel.size(-1) // 2, groups
                            =self.out_channels)
199
200     def compute_activation(self, x):
201         x = symmetric_pad(x, self.padding)
202         w = self.weight.abs()
203         if self.w_nrm: w = normalize(w, dim=(1, 2, 3))
204         if self.presynaptic_weights: w = self.
               compute_presynaptic_competition(w)
205         # y_depthwise = self.act(self.apply_weights(x, w))
```

```
206         # For cosine similarity activation if cosine is to be used
                for next layer
207         y_depthwise = self.cosine(x,w)
208         return x,y_depthwise, w
209
210     def forward(self, x):
211         x,y_depthwise, w = self.compute_activation(x)
212         if self.kernel !=1:
213             y_depthwise = self.apply_surround_modulation(y_depthwise
                    )
214         if self.training:
215             self.compute_update(x, y_depthwise, w)
216         return y_depthwise
217
218     def compute_update(self, x, y, weight):
219         if self.mode == self.MODE_BASIC_HEBBIAN:
220             update = self.update_basic_hebbian(x, y, weight)
221         elif self.mode == self.MODE_HARDWT:
222             update = self.update_hardwt(x, y, weight)
223         elif self.mode == self.MODE_SOFTWTA:
224             update = self.update_softwta(x, y, weight)
225         elif self.mode == self.MODE_ANTIHARDWT:
226             update = self.update_antihardwt(x, y, weight)
227         elif self.mode == self.MODE_BCM:
228             update = self.update_bcm(x, y, weight)
229         elif self.mode == self.MODE_TEMPORAL_COMPETITION:
230             update = self.update_temporal_competition(x, y, weight)
231         elif self.mode == self.MODE_ADAPTIVE_THRESHOLD:
232             update = self.update_adaptive_threshold(x, y, weight)
233         else:
234             raise NotImplementedError(f"Learning mode {self.mode}
                    unavailable for {self.__class__.__name__} layer")
235
236         # Weight Normalization and added to weight change buffer
237         update.div_(torch.abs(update).amax() + 1e-30)
238         self.delta_w += update
239
240     def update_basic_hebbian(self, x, y, weight):
241         yx = self.compute_yx(x, y)
242         y_sum = y.sum(dim=(0, 2, 3)).view(self.in_channels, 1, 1, 1)
243         yw = y_sum * weight
244         return yx - yw
245
246     def update_hardwt(self, x, y, weight):
247         y_wta = y * self.compute_wta_mask(y)
248         yx = self.compute_yx(x, y_wta)
249         yu = torch.sum(y_wta, dim=(0, 2, 3)).view(self.in_channels,
                1, 1, 1)
250         return yx - yu * weight
251
252     def update_softwta(self, x, y, weight):
253         softwta_activs = self.compute_softwta_activations(y)
254         yx = self.compute_yx(x, softwta_activs)
255         yu = torch.sum(torch.mul(softwta_activs, y), dim=(0, 2, 3)).
                view(self.in_channels, 1, 1, 1)
256         return yx - yu * weight
257
258     def update_bcm(self, x, y, weight):
```

```
259         y_wta = y * self.compute_wta_mask(y)
260         y_squared = y_wta.pow(2).mean(dim=(0, 2, 3))
261         self.theta.data = (1 - self.theta_decay) * self.theta + self
                .theta_decay * y_squared
262         y_minus_theta = y_wta - self.theta.view(1, -1, 1, 1)
263         bcm_factor = y_wta * y_minus_theta
264         yx = self.compute_yx(x, bcm_factor)
265         return yx.view(weight.shape)
266 
267     def update_temporal_competition(self, x, y, weight):
268         batch_size, out_channels, height_out, width_out = y.shape
269         self.update_activation_history(y)
270         temporal_winners = self.compute_temporal_winners(y)
271         y_winners = temporal_winners * y
272         y_winners = self.apply_competition(y_winners, batch_size,
                out_channels)
273         yx = self.compute_yx(x, y_winners)
274         y_sum = y_winners.sum(dim=(0, 2, 3)).view(self.in_channels,
                1, 1, 1)
275         update = yx - y_sum * weight
276         return update
277 
278     def update_adaptive_threshold(self, x, y, weight):
279         batch_size, out_channels, height_out, width_out = y.shape
280         similarities = F.conv2d(x, weight, stride=self.stride,
                padding=self.padding, groups=self.groups)
281         similarities = similarities / (torch.norm(weight.view(
                out_channels, -1), dim=1).view(1, -1, 1, 1) + 1e-10)
282         threshold = self.compute_adaptive_threshold(similarities)
283         winners = (similarities > threshold).float()
284         y_winners = winners * similarities
285         y_winners = self.apply_competition(y_winners, batch_size,
                out_channels)
286         yx = self.compute_yx(x, y_winners)
287         y_sum = y_winners.sum(dim=(0, 2, 3)).view(self.in_channels,
                1, 1, 1)
288         update = yx - y_sum * weight
289         return update
290 
291     def compute_yx(self, x, y):
292         yx = F.conv2d(x.transpose(0, 1), y.transpose(0, 1), padding
                =0,
293                       stride=self.dilation, dilation=self.stride).
                          transpose(0, 1)
294         yx = yx.diagonal(dim1=0, dim2=1).permute(2, 0, 1).unsqueeze
                (1)
295         return yx
296 
297     def compute_wta_mask(self, y):
298         batch_size, out_channels, height_out, width_out = y.shape
299         # WTA competition within each channel
300         y_flat = y.view(batch_size, out_channels, -1)
301         win_neurons = torch.argmax(y_flat, dim=2)
302         wta_mask = F.one_hot(win_neurons, num_classes=height_out *
                width_out).float()
303         return wta_mask.view(batch_size, out_channels, height_out,
                width_out)
304 
```

```
305     def compute_softwta_activations(self, y):
306         # Competition and anti-Hebbian learning for y_depthwise
307         batch_size, in_channels, height_depthwise, width_depthwise =
                y.shape
308         # Reshape to apply softmax within each channel
309         y_depthwise_reshaped = y.view(batch_size, in_channels, -1)
310         # Apply softmax within each channel
311         flat_softwta_activs_depthwise = torch.softmax(self.t_invert
               * y_depthwise_reshaped, dim=2)
312         # Turn all postsynaptic activations into anti-Hebbian
313         flat_softwta_activs_depthwise = -
               flat_softwta_activs_depthwise
314         # Find winners within each channel
315         win_neurons_depthwise = torch.argmax(y_depthwise_reshaped,
               dim=2)
316         # Create a mask to flip the sign of winning neurons
317         mask = torch.zeros_like(flat_softwta_activs_depthwise)
318         mask.scatter_(2, win_neurons_depthwise.unsqueeze(2), 1)
319         # Flip the sign of winning neurons
320         flat_softwta_activs_depthwise =
               flat_softwta_activs_depthwise * (1 - 2 * mask)
321         # Reshape back to original shape
322         return flat_softwta_activs_depthwise.view(batch_size,
               in_channels, height_depthwise,
323                                                                 width_dept
                                                                    )
                                                                    )
324
325     def update_activation_history(self, y):
326         if self.activation_history is None:
327             self.activation_history = y.detach().clone()
328         else:
329             self.activation_history = torch.cat([self.
                   activation_history, y.detach()], dim=0)
330             if self.activation_history.size(0) > self.
                   temporal_window:
331                 self.activation_history = self.activation_history[-
                       self.temporal_window:]
332
333     def compute_temporal_winners(self, y):
334         batch_size, out_channels, height_out, width_out = y.shape
335         history_spatial = self.activation_history.view(-1,
               out_channels, height_out, width_out)
336         median_activations = torch.median(history_spatial, dim=0)[0]
337         temporal_threshold = torch.mean(median_activations, dim=(1,
               2), keepdim=True)
338         return (median_activations > temporal_threshold).float()
339
340     def compute_adaptive_threshold(self, similarities):
341         mean_sim = similarities.mean(dim=(2, 3), keepdim=True)
342         std_sim = similarities.std(dim=(2, 3), keepdim=True)
343         return mean_sim + self.competition_k * std_sim
344
345     def apply_competition(self, y, batch_size, out_channels):
346         if self.mode in [self.MODE_TEMPORAL_COMPETITION, self.
               MODE_ADAPTIVE_THRESHOLD]:
347             if self.competition_type == 'hard':
348                 y = y.view(batch_size, out_channels, -1)
```

```
                top_k_indices = torch.topk(y, self.top_k, dim=2,
                    largest=True, sorted=False).indices
                y_compete = torch.zeros_like(y)
                y_compete.scatter_(2, top_k_indices, y.gather(2,
                    top_k_indices))
                return y_compete.view_as(y)
            elif self.competition_type == 'soft':
                return torch.softmax(self.t_invert * y.view(
                    batch_size, out_channels, -1), dim=2).view_as(y)
        return y

    @torch.no_grad()
    def local_update(self):
        new_weight = self.weight + 0.1 * self.alpha * self.delta_w
        self.weight.copy_(new_weight.abs())
        self.delta_w.zero_()
```

A.2.6 Model_hebb.py

```
import torch
import torch.nn as nn
import torch.nn.functional as F

from hebb import HebbianConv2d
# To use Dale Principle, manuallly change this line
# from hebb_abs import HebbianConv2d

import matplotlib.pyplot as plt
import numpy as np
import wandb
import seaborn as sns

"""
File handles the creation of Hebbian models
"""

torch.manual_seed(0)
DEFAULT_HEBB_PARAMS = {'mode': HebbianConv2d.MODE_SOFTWTA, 'w_nrm':
    True, 'k': 50, 'act': nn.Identity(), 'alpha': 1.}

# Activation function, as described in https://github.com/
    NeuromorphicComputing/SoftHebb
class Triangle(nn.Module):
    def __init__(self, power: float = 1, inplace: bool = True):
        super(Triangle, self).__init__()
        self.inplace = inplace
        self.power = power

    def forward(self, input: torch.Tensor) -> torch.Tensor:
        input = input - torch.mean(input.data, axis=1, keepdims=True
            )
        return F.relu(input, inplace=self.inplace) ** self.power

class Net_Hebbian(nn.Module):
    def __init__(self, hebb_params=None, version="softhebb"):
        super(Net_Hebbian, self).__init__()
        self.hebb_params = hebb_params or DEFAULT_HEBB_PARAMS
        self.version = version
```

```
        self._build_network()

    def _build_network(self):
        if self.version == "softhebb":
            self._build_softhebb_network()
        elif self.version == "hardhebb":
            self._build_hardhebb_network()
        elif self.version == "lagani":
            self._build_lagani_network()
        elif self.version == "lagani_short":
            self._build_lagani_short_network()
        else:
            raise ValueError(f"Unknown version: {self.version}")

    # Architecture equivalent to SoftHebb research
    def _build_softhebb_network(self):
        # Layer 1
        self.bn1 = nn.BatchNorm2d(3, affine=False)
        self.conv1 = HebbianConv2d(in_channels=3, out_channels=96,
           kernel_size=5, stride=1, **self.hebb_params,
                                   padding=2, t_invert=1)
        self.pool1 = nn.MaxPool2d(kernel_size=4, stride=2, padding
           =1)
        self.activ1 = Triangle(power=0.7)

        # Layer 2
        self.bn2 = nn.BatchNorm2d(96, affine=False)
        self.conv2 = HebbianConv2d(in_channels=96, out_channels=384,
            kernel_size=3, stride=1, **self.hebb_params,
                                       t_invert=0.65, padding=1)
        self.pool2 = nn.MaxPool2d(kernel_size=4, stride=2, padding
           =1)
        self.activ2 = Triangle(power=1.4)

        # Layer 3
        self.bn3 = nn.BatchNorm2d(384, affine=False)
        self.conv3 = HebbianConv2d(in_channels=384, out_channels
           =1536, kernel_size=3, stride=1, **self.hebb_params,
                                       t_invert=0.25, padding=1)
        self.pool3 = nn.AvgPool2d(kernel_size=2, stride=2, padding
           =0)
        self.activ3 = Triangle(power=1.)

        # Output layers
        self.flatten = nn.Flatten()
        self.fc1 = nn.Linear(24576, 10)
        self.fc1.weight.data = 0.11048543456039805 * torch.rand(10,
           24576)
        self.dropout = nn.Dropout(0.5)

    # Architecture equivalent to SoftHebb research, but without
       padding
    def _build_hardhebb_network(self): # Hybrid: Similar to Lagani
       but with softhebb number of filters and layers
        # Layer 1
        self.bn1 = nn.BatchNorm2d(3, affine=False)
        self.conv1 = HebbianConv2d(in_channels=3, out_channels=96,
           kernel_size=5, stride=1, **self.hebb_params,
```

```
                                    padding=0, t_invert=1)
        self.pool1 = nn.MaxPool2d(kernel_size=2, stride=2, padding
            =0)
        self.activ1 = Triangle(power=0.7)

        # Layer 2
        self.bn2 = nn.BatchNorm2d(96, affine=False)
        self.conv2 = HebbianConv2d(in_channels=96, out_channels=384,
             kernel_size=3, stride=1, **self.hebb_params,
                                    t_invert=0.65, padding=0)
        self.activ2 = Triangle(power=1.4)

        # Layer 3
        self.bn3 = nn.BatchNorm2d(384, affine=False)
        self.conv3 = HebbianConv2d(in_channels=384, out_channels
            =1536, kernel_size=3, stride=1,
                                    **self.hebb_params, t_invert
                                        =0.25, padding=0)
        self.pool3 = nn.AvgPool2d(kernel_size=2, stride=2, padding
            =0)
        self.activ3 = Triangle(power=1.)

        # Output layers
        self.flatten = nn.Flatten()
        self.fc1 = nn.Linear(38400, 10)
        self.fc1.weight.data = 0.11048543456039805 * torch.rand(10,
            38400)
        self.dropout = nn.Dropout(0.5)

    # Architecture equivalent to Lagani 3-layer research
    def _build_lagani_short_network(self):
        # Layer 1
        self.bn1 = nn.BatchNorm2d(3, affine=False)
        self.conv1 = HebbianConv2d(in_channels=3, out_channels=96,
            kernel_size=5, stride=1, **self.hebb_params,
                                    padding=0, t_invert=1)
        self.pool1 = nn.MaxPool2d(kernel_size=2, stride=2, padding
            =0)
        self.activ1 = Triangle(power=1.)

        # Layer 2
        self.bn2 = nn.BatchNorm2d(96, affine=False)
        self.conv2 = HebbianConv2d(in_channels=96, out_channels=128,
             kernel_size=3, stride=1, **self.hebb_params,
                                    t_invert=0.65, padding=0)
        # self.pool2 = nn.MaxPool2d(kernel_size=2, stride=2, padding
            =0)
        self.activ2 = Triangle(power=1.)

        # Layer 3
        self.bn3 = nn.BatchNorm2d(128, affine=False)
        self.conv3 = HebbianConv2d(in_channels=128, out_channels
            =192, kernel_size=3, stride=1, **self.hebb_params,
                                    t_invert=0.25, padding=0)
        self.pool3 = nn.AvgPool2d(kernel_size=2, stride=2, padding
            =0)
        self.activ3 = Triangle(power=1.)

```

```
131         # Output layers
132         self.flatten = nn.Flatten()
133         self.fc1 = nn.Linear(4800, 10)
134         self.fc1.weight.data = 0.11048543456039805 * torch.rand(10,
                4800)
135         self.dropout = nn.Dropout(0.5)
136 
137     # Architecture equivalent to Lagani 4-layer research
138     def _build_lagani_network(self):
139         # Layer 1
140         self.bn1 = nn.BatchNorm2d(3, affine=False)
141         self.conv1 = HebbianConv2d(in_channels=3, out_channels=96,
                kernel_size=5, stride=1, **self.hebb_params,
142                                    padding=0, t_invert=1)
143         self.pool1 = nn.MaxPool2d(kernel_size=2, stride=2, padding
                =0)
144         self.activ1 = Triangle(power=1.)
145 
146         # Layer 2
147         self.bn2 = nn.BatchNorm2d(96, affine=False)
148         self.conv2 = HebbianConv2d(in_channels=96, out_channels=128,
                 kernel_size=3, stride=1, **self.hebb_params,
149                                    t_invert=0.65, padding=0)
150         self.activ2 = Triangle(power=1.)
151 
152         # Layer 3
153         self.bn3 = nn.BatchNorm2d(128, affine=False)
154         self.conv3 = HebbianConv2d(in_channels=128, out_channels
                =192, kernel_size=3, stride=1, **self.hebb_params,
155                                    t_invert=0.25, padding=0)
156         self.pool3 = nn.AvgPool2d(kernel_size=2, stride=2, padding
                =0)
157         self.activ3 = Triangle(power=1.)
158 
159         # Layer 4
160         self.bn4 = nn.BatchNorm2d(192, affine=False)
161         self.conv4 = HebbianConv2d(in_channels=192, out_channels
                =256, kernel_size=3, stride=1, **self.hebb_params,
162                                    t_invert=0.25, padding=0)
163         self.activ4 = Triangle(power=1.)
164 
165         # Output layers
166         self.flatten = nn.Flatten()
167         self.fc1 = nn.Linear(2304, 10)
168         self.fc1.weight.data = 0.11048543456039805 * torch.rand(10,
                2304)
169         self.dropout = nn.Dropout(0.5)
170 
171     def forward_features(self, x):
172         x = self.pool1(self.activ1(self.conv1(self.bn1(x))))
173         return x
174 
175     def features_extract(self, x):
176         x = self.forward_features(x)
177         if self.version == "lagani":
178             x = self.activ2(self.conv2(self.bn2(x)))
179             x = self.pool3(self.activ3(self.conv3(self.bn3(x))))
180             x = self.activ4(self.conv4(self.bn4(x)))
```

```
        elif self.version == "lagani_short":
            x = self.activ2(self.conv2(self.bn2(x)))
            x = self.pool3(self.activ3(self.conv3(self.bn3(x))))
        elif self.version == "hardhebb":
            x = self.activ2(self.conv2(self.bn2(x)))
            x = self.pool3(self.activ3(self.conv3(self.bn3(x))))
        elif self.version == "softhebb":
            x = self.pool2(self.activ2(self.conv2(self.bn2(x))))
            x = self.pool3(self.activ3(self.conv3(self.bn3(x))))
        return x

    def forward(self, x):
        x = self.features_extract(x)
        x = self.flatten(x)
        x = self.fc1(self.dropout(x))
        return x

    # Plot neurons/filter of a target layer
    def plot_grid(self, tensor, path, num_rows=5, num_cols=5,
        layer_name=""):
        # Ensure we're working with the first 25 filters (or less if
            there are fewer)
        excitatory = tensor[:20]
        inhibitory = tensor[-5:]
        # Symmetric normalization for excitatory weights
        max_abs_exc = torch.max(torch.abs(excitatory))
        norm_exc = excitatory / (max_abs_exc + 1e-8)
        # Symmetric normalization for inhibitory weights
        max_abs_inh = torch.max(torch.abs(inhibitory))
        norm_inh = inhibitory / (max_abs_inh + 1e-8)
        tensor = torch.cat((norm_exc, norm_inh))
        # Normalize the tensor
        # Move to CPU and convert to numpy
        tensor = tensor.cpu().detach().numpy()

        if tensor.shape[2] == 1 and tensor.shape[3] == 1:  # 1x1
            convolution case
            out_channels, in_channels = tensor.shape[:2]
            fig = plt.figure(figsize=(14, 10))
            # Create a gridspec for the layout
            gs = fig.add_gridspec(2, 2, width_ratios=[20, 1],
                height_ratios=[1, 3],
                                  left=0.1, right=0.9, bottom=0.1,
                                      top=0.9, wspace=0.05, hspace
                                      =0.2)
            ax1 = fig.add_subplot(gs[0, 0])
            ax2 = fig.add_subplot(gs[1, 0])
            cbar_ax = fig.add_subplot(gs[:, 1])
            # Bar plot for average weights per filter
            avg_weights = tensor.mean(axis=(1, 2, 3))
            norm = plt.Normalize(vmin=avg_weights.min(), vmax=
                avg_weights.max())
            im1 = ax1.bar(range(out_channels), avg_weights, color=
                plt.cm.RdYlGn(norm(avg_weights)))
            ax1.set_xlabel('Filter Index')
            ax1.set_ylabel('Average Weight')
            ax1.set_title(f'Average Weights for 1x1 Kernels in {
                layer_name}')
```

```
            ax1.axhline(y=0, color='black', linestyle='--',
                linewidth=0.5)
            # Heatmap for detailed weight distribution
            im2 = ax2.imshow(tensor.reshape(out_channels,
                in_channels), cmap='RdYlGn', aspect='auto', norm=norm
                )
            ax2.set_xlabel('Input Channel')
            ax2.set_ylabel('Output Channel (Filter)')
            ax2.set_title('Detailed Weight Distribution')
            # Add colorbar to the right of both subplots
            fig.colorbar(im2, cax=cbar_ax, label='Normalized Weight
                Value')

        else:
            fig, axes = plt.subplots(num_rows, num_cols, figsize
                =(15, 10))
            fig.suptitle(f'First 25 Filters of {layer_name}')
            for i, ax in enumerate(axes.flat):
                if i < tensor.shape[0]:
                    filter_img = tensor[i]
                    # Handle different filter shapes
                    if filter_img.shape[0] == 3:  # RGB filter (3, H
                        , W)
                        filter_img = np.transpose(filter_img, (1, 2,
                            0))
                        filter_img = (filter_img - filter_img.min())
                            / (filter_img.max() - filter_img.min() +
                            1e-8)
                    elif filter_img.shape[0] == 1:  # Grayscale
                        filter (1, H, W)
                        filter_img = filter_img.squeeze()
                    else:  # Multi-channel filter (C, H, W), take
                        mean across channels
                        filter_img = np.mean(filter_img, axis=0)
                    ax.imshow(filter_img, cmap='viridis' if
                        filter_img.ndim == 2 else None)
                    ax.set_title(f'Filter {i + 1}')
                ax.axis('off')

        if path:
            fig.savefig(path, bbox_inches='tight')
        wandb.log({f'{layer_name} filters': wandb.Image(fig)})
        plt.close(fig)


    def visualize_filters(self, layer_name='conv1', save_path=None):
        weights = getattr(self, layer_name).weight.data
        self.plot_grid(weights, save_path, layer_name=layer_name)
```

A.2.7 Model_depthwise

```
import torch
import torch.nn as nn
import torch.nn.functional as F

from hebb import HebbianConv2d
from hebb_depthwise import HebbianDepthConv2d

```

```
# from hebb_abs import HebbianConv2d
# from hebb_abs_depthwise import HebbianDepthConv2d

import matplotlib.pyplot as plt
import numpy as np
import wandb
import seaborn as sns

torch.manual_seed(0)
DEFAULT_HEBB_PARAMS = {'mode': HebbianConv2d.MODE_SOFTWTA, 'w_nrm': True, 'k': 50, 'act': nn.Identity(), 'alpha': 1.}


class Triangle(nn.Module):
    def __init__(self, power: float = 1, inplace: bool = True):
        super(Triangle, self).__init__()
        self.inplace = inplace
        self.power = power

    def forward(self, input: torch.Tensor) -> torch.Tensor:
        input = input - torch.mean(input.data, axis=1, keepdims=True)
        return F.relu(input, inplace=self.inplace) ** self.power

class Net_Depthwise(nn.Module):
    def __init__(self, hebb_params=None, version="softhebb"):
        super(Net_Depthwise, self).__init__()
        self.hebb_params = hebb_params or DEFAULT_HEBB_PARAMS
        self.version = version
        self._build_network()

    def _build_network(self):
        if self.version == "softhebb":
            self._build_softhebb_network()
        elif self.version == "hardhebb":
            self._build_hardhebb_network()
        elif self.version == "lagani":
            self._build_lagani_network()
        else:
            raise ValueError(f"Unknown version: {self.version}")

    def _build_softhebb_network(self):
        # Layer 1
        self.bn1 = nn.BatchNorm2d(3, affine=False)
        self.conv1 = HebbianConv2d(in_channels=3, out_channels=96, kernel_size=5, stride=1, **self.hebb_params,
                                   padding=2, t_invert=1)
        self.pool1 = nn.MaxPool2d(kernel_size=4, stride=2, padding=1)
        self.activ1 = Triangle(power=0.7)

        # Layer 2
        self.bn2 = nn.BatchNorm2d(96, affine=False)
        self.conv2 = HebbianDepthConv2d(in_channels=96, out_channels=96, kernel_size=3, stride=1, **self.hebb_params,
                                        t_invert=0.65, padding=1)
        self.bn_point2 = nn.BatchNorm2d(96, affine=False)
```

```
60          self.conv_point2 = HebbianConv2d(in_channels=96,
                out_channels=384, kernel_size=1, stride=1, **self.
                hebb_params,
61                                           t_invert=0.65, padding=0)
62          self.pool2 = nn.MaxPool2d(kernel_size=4, stride=2, padding
                =1)
63          self.activ2 = Triangle(power=1.4)
64
65          # Layer 3
66          self.bn3 = nn.BatchNorm2d(384, affine=False)
67          self.conv3 = HebbianDepthConv2d(in_channels=384,
                out_channels=384, kernel_size=3, stride=1, **self.
                hebb_params,
68                                          t_invert=0.25, padding=1)
69          self.bn_point3 = nn.BatchNorm2d(384, affine=False)
70          self.conv_point3 = HebbianConv2d(in_channels=384,
                out_channels=1536, kernel_size=1, stride=1, **self.
                hebb_params,
71                                           t_invert=0.25, padding=0)
72          self.pool3 = nn.AvgPool2d(kernel_size=2, stride=2, padding
                =0)
73          self.activ3 = Triangle(power=1.)
74
75          # Output layers
76          self.flatten = nn.Flatten()
77          self.fc1 = nn.Linear(24576, 10)
78          self.fc1.weight.data = 0.11048543456039805 * torch.rand(10,
                24576)
79          self.dropout = nn.Dropout(0.5)
80
81      def _build_hardhebb_network(self):
82          # Layer 1
83          self.bn1 = nn.BatchNorm2d(3, affine=False)
84          self.conv1 = HebbianConv2d(in_channels=3, out_channels=96,
                kernel_size=5, stride=1, **self.hebb_params,
85                                     padding=0, t_invert=1)
86          self.pool1 = nn.MaxPool2d(kernel_size=2, stride=2, padding
                =0)
87          self.activ1 = Triangle(power=0.7)
88
89          # Layer 2
90          self.bn2 = nn.BatchNorm2d(96, affine=False)
91          self.conv2 = HebbianDepthConv2d(in_channels=96, out_channels
                =96, kernel_size=3, stride=1, **self.hebb_params,
92                                          t_invert=0.65, padding=0)
93          self.bn_point2 = nn.BatchNorm2d(96, affine=False)
94          self.conv_point2 = HebbianConv2d(in_channels=96,
                out_channels=384, kernel_size=1, stride=1,
95                                           **self.hebb_params,
                                                t_invert=0.65, padding
                                                =0)
96          self.activ2 = Triangle(power=1.4)
97
98          # Layer 3
99          self.bn3 = nn.BatchNorm2d(384, affine=False)
100         self.conv3 = HebbianDepthConv2d(in_channels=384,
                out_channels=384, kernel_size=3, stride=1,
```

```
                                        **self.hebb_params, t_invert=0.25, padding=0)
        self.bn_point3 = nn.BatchNorm2d(384, affine=False)
        self.conv_point3 = HebbianConv2d(in_channels=384, out_channels=1536, kernel_size=1, stride=1,
                                         **self.hebb_params, t_invert=0.25, padding=0)
        self.pool3 = nn.AvgPool2d(kernel_size=2, stride=2, padding=0)
        self.activ3 = Triangle(power=1.)

        # Output layers
        self.flatten = nn.Flatten()
        self.fc1 = nn.Linear(38400, 10)
        self.fc1.weight.data = 0.11048543456039805 * torch.rand(10, 38400)
        self.dropout = nn.Dropout(0.5)

    def _build_lagani_network(self):
        # Layer 1
        self.bn1 = nn.BatchNorm2d(3, affine=False)
        self.conv1 = HebbianConv2d(in_channels=3, out_channels=96, kernel_size=5, stride=1, **self.hebb_params,
                                   padding=0, t_invert=1)
        self.pool1 = nn.MaxPool2d(kernel_size=2, stride=2, padding=0)
        self.activ1 = Triangle(power=1.)

        # Layer 2
        self.bn2 = nn.BatchNorm2d(96, affine=False)
        self.conv2 = HebbianDepthConv2d(in_channels=96, out_channels=96, kernel_size=3, stride=1, **self.hebb_params,
                                        t_invert=0.65, padding=0)
        self.bn_point2 = nn.BatchNorm2d(96, affine=False)
        self.conv_point2 = HebbianConv2d(in_channels=96, out_channels=128, kernel_size=1, stride=1, **self.hebb_params,
                                         t_invert=0.65, padding=0)
        self.pool2 = nn.MaxPool2d(kernel_size=2, stride=2, padding=0)
        self.activ2 = Triangle(power=1.)

        # Layer 3
        self.bn3 = nn.BatchNorm2d(128, affine=False)
        self.conv3 = HebbianDepthConv2d(in_channels=128, out_channels=128, kernel_size=3, stride=1, **self.hebb_params,
                                        t_invert=0.25, padding=0)
        self.bn_point3 = nn.BatchNorm2d(128, affine=False)
        self.conv_point3 = HebbianConv2d(in_channels=128, out_channels=192, kernel_size=1, stride=1, **self.hebb_params,
                                         t_invert=0.25, padding=0)
        self.pool3 = nn.AvgPool2d(kernel_size=2, stride=2, padding=0)
        self.activ3 = Triangle(power=1.)

```

```
142         # Layer 4
143         self.bn4 = nn.BatchNorm2d(192, affine=False)
144         self.conv4 = HebbianDepthConv2d(in_channels=192,
                out_channels=192, kernel_size=3, stride=1, **self.
                hebb_params,
145                                         t_invert=0.25, padding=0)
146         self.bn_point4 = nn.BatchNorm2d(192, affine=False)
147         self.conv_point4 = HebbianConv2d(in_channels=192,
                out_channels=256, kernel_size=1, stride=1, **self.
                hebb_params,
148                                          t_invert=0.25, padding=0)
149         self.activ4 = Triangle(power=1.)
150
151         # Output layers
152         self.flatten = nn.Flatten()
153         self.fc1 = nn.Linear(1728, 10)
154         self.fc1.weight.data = 0.11048543456039805 * torch.rand(10,
                1728)
155         self.dropout = nn.Dropout(0.5)
156
157     def forward_features(self, x):
158         x = self.pool1(self.activ1(self.conv1(self.bn1(x))))
159         return x
160
161     def features_extract(self, x):
162         x = self.forward_features(x)
163         if self.version == "lagani":
164             x = self.pool2(self.activ2(self.conv_point2(self.
                    bn_point2(self.conv2(self.bn2(x))))))
165             x = self.pool3(self.activ3(self.conv_point3(self.
                    bn_point3(self.conv3(self.bn3(x))))))
166             x = self.activ4(self.conv_point4(self.bn_point4(self.
                    conv4(self.bn4(x)))))
167         elif self.version == "hardhebb":
168             x = self.activ2(self.conv_point2(self.bn_point2(self.
                    conv2(self.bn2(x)))))
169             x = self.pool3(self.activ3(self.conv_point3(self.
                    bn_point3(self.conv3(self.bn3(x))))))
170         elif self.version == "softhebb":
171             x = self.pool2(self.activ2(self.conv_point2(self.
                    bn_point2(self.conv2(self.bn2(x))))))
172             x = self.pool3(self.activ3(self.conv_point3(self.
                    bn_point3(self.conv3(self.bn3(x))))))
173         return x
174
175     def forward(self, x):
176         x = self.features_extract(x)
177         x = self.flatten(x)
178         x = self.fc1(self.dropout(x))
179         return x
180
181     def plot_grid(self, tensor, path, num_rows=5, num_cols=5,
            layer_name=""):
182         # Ensure we're working with the first 25 filters (or less if
                there are fewer)
183         excitatory = tensor[:20]
184         inhibitory = tensor[-5:]
185         # Symmetric normalization for excitatory weights
```

```
        max_abs_exc = torch.max(torch.abs(excitatory))
        norm_exc = excitatory / (max_abs_exc + 1e-8)
        # Symmetric normalization for inhibitory weights
        max_abs_inh = torch.max(torch.abs(inhibitory))
        norm_inh = inhibitory / (max_abs_inh + 1e-8)
        tensor = torch.cat((norm_exc, norm_inh))
        # Move to CPU and convert to numpy
        tensor = tensor.cpu().detach().numpy()

        if tensor.shape[2] == 1 and tensor.shape[3] == 1:  # 1x1 convolution case
            out_channels, in_channels = tensor.shape[:2]
            fig = plt.figure(figsize=(14, 10))
            # Create a gridspec for the layout
            gs = fig.add_gridspec(2, 2, width_ratios=[20, 1], height_ratios=[1, 3],
                                  left=0.1, right=0.9, bottom=0.1, top=0.9, wspace=0.05, hspace=0.2)
            ax1 = fig.add_subplot(gs[0, 0])
            ax2 = fig.add_subplot(gs[1, 0])
            cbar_ax = fig.add_subplot(gs[:, 1])
            # Bar plot for average weights per filter
            avg_weights = tensor.mean(axis=(1, 2, 3))
            norm = plt.Normalize(vmin=avg_weights.min(), vmax=avg_weights.max())
            im1 = ax1.bar(range(out_channels), avg_weights, color=plt.cm.RdYlGn(norm(avg_weights)))
            ax1.set_xlabel('Filter Index')
            ax1.set_ylabel('Average Weight')
            ax1.set_title(f'Average Weights for 1x1 Kernels in {layer_name}')
            ax1.axhline(y=0, color='black', linestyle='--', linewidth=0.5)
            # Heatmap for detailed weight distribution
            im2 = ax2.imshow(tensor.reshape(out_channels, in_channels), cmap='RdYlGn', aspect='auto', norm=norm)
            ax2.set_xlabel('Input Channel')
            ax2.set_ylabel('Output Channel (Filter)')
            ax2.set_title('Detailed Weight Distribution')
            # Add colorbar to the right of both subplots
            fig.colorbar(im2, cax=cbar_ax, label='Normalized Weight Value')

        else:
            fig, axes = plt.subplots(num_rows, num_cols, figsize=(15, 10))
            fig.suptitle(f'First 25 Filters of {layer_name}')
            for i, ax in enumerate(axes.flat):
                if i < tensor.shape[0]:
                    filter_img = tensor[i]
                    # Handle different filter shapes
                    if filter_img.shape[0] == 3:  # RGB filter (3, H, W)
                        filter_img = np.transpose(filter_img, (1, 2, 0))
```

```
                        filter_img = (filter_img - filter_img.min())
                            / (filter_img.max() - filter_img.min() +
                            1e-8)
                    elif filter_img.shape[0] == 1:  # Grayscale
                        filter (1, H, W)
                        filter_img = filter_img.squeeze()
                    else:  # Multi-channel filter (C, H, W), take
                        mean across channels
                        filter_img = np.mean(filter_img, axis=0)
                    ax.imshow(filter_img, cmap='viridis' if
                        filter_img.ndim == 2 else None)
                    ax.set_title(f'Filter {i + 1}')
                ax.axis('off')

        if path:
            fig.savefig(path, bbox_inches='tight')
        wandb.log({f'{layer_name} filters': wandb.Image(fig)})
        plt.close(fig)

    def visualize_filters(self, layer_name='conv1', save_path=None):
        weights = getattr(self, layer_name).weight.data
        self.plot_grid(weights, save_path, layer_name=layer_name)
```

A.2.8 Model_residual.py

```
import torch
import torch.nn as nn
import torch.nn.functional as F

from hebb import HebbianConv2d
from hebb_depthwise import HebbianDepthConv2d

import matplotlib.pyplot as plt
import numpy as np
import wandb
import seaborn as sns

"""
File handles the creation of Hebbian Residual models
Similar to mode_hebb.py, refer to this file for further explanations
"""

torch.manual_seed(0)
default_hebb_params = {'mode': HebbianConv2d.MODE_SOFTWTA, 'w_nrm':
    True, 'k': 50, 'act': nn.Identity(), 'alpha': 1.}


class Triangle(nn.Module):
    def __init__(self, power: float = 1, inplace: bool = True):
        super(Triangle, self).__init__()
        self.inplace = inplace
        self.power = power

    def forward(self, input: torch.Tensor) -> torch.Tensor:
        input = input - torch.mean(input.data, axis=1, keepdims=True
            )
        return F.relu(input, inplace=self.inplace) ** self.power

```

```
class LongSkipConnection(nn.Module):
    def __init__(self, in_channels, out_channels, spatial_change=1, hebb_params=None):
        super(LongSkipConnection, self).__init__()
        if hebb_params is None:
            hebb_params = default_hebb_params

        self.conv = HebbianConv2d(in_channels, out_channels, kernel_size=1, stride=1, **hebb_params)
        self.bn = nn.BatchNorm2d(out_channels, affine=False)
        self.spatial_change = spatial_change

    def forward(self, x):
        x = self.conv(x)
        x = self.bn(x)
        if self.spatial_change != 1:
            x = F.adaptive_avg_pool2d(x, output_size=x.size()[2:] // self.spatial_change)
        return x


class HebbianResidualBlock(nn.Module):
    def __init__(self, in_channels, out_channels, kernel_size, stride=1, hebb_params=None, t_invert=1., expansion_factor=6, act=1.):
        super(HebbianResidualBlock, self).__init__()
        if hebb_params is None:
            hebb_params = default_hebb_params
        # Calculate padding to maintain spatial dimensions
        # Doubt regarding when additional padding is required
        padding = (kernel_size - 1) // 2
        hidden_dim = in_channels * expansion_factor
        self.bn1 = nn.BatchNorm2d(in_channels, affine=False)
        self.conv1 = HebbianConv2d(in_channels, hidden_dim, kernel_size=1, stride=1, **hebb_params, t_invert=t_invert,
                                   padding=0)
        self.bn2 = nn.BatchNorm2d(hidden_dim, affine=False)
        self.conv2 = HebbianDepthConv2d(hidden_dim, hidden_dim, kernel_size, stride=1, **hebb_params,
                                        t_invert=t_invert, padding=padding)
        self.bn3 = nn.BatchNorm2d(hidden_dim, affine=False)
        self.conv3 = HebbianConv2d(hidden_dim, out_channels, kernel_size=1, stride=1, **hebb_params, t_invert=t_invert,
                                   padding=0)
        self.activ = Triangle(power=act)
        self.shortcut = nn.Sequential()
        if stride != 1 or in_channels != out_channels:
            self.shortcut = nn.Sequential(
            nn.BatchNorm2d(in_channels, affine=False),
            HebbianConv2d(in_channels, out_channels, kernel_size=1, stride=1, **hebb_params, padding=0)
            )

    def forward(self, x):
        residual = x
```

```
78          out = self.activ(self.conv1(self.bn1(x)))
79          out = self.activ(self.conv2(self.bn2(out)))
80          out = self.conv3(self.bn3(out))
81          out += self.shortcut(residual)
82          return self.activ(out)
83
84  class Net_Depthwise_Residual(nn.Module):
85      def __init__(self, hebb_params=None):
86          super(Net_Depthwise_Residual, self).__init__()
87
88          if hebb_params is None:
89              hebb_params = default_hebb_params
90
91          self.bn1 = nn.BatchNorm2d(3, affine=False)
92          self.conv1 = HebbianConv2d(in_channels=3, out_channels=96,
                kernel_size=5, stride=1, **hebb_params, padding=2)
93          self.pool1 = nn.MaxPool2d(kernel_size=4, stride=2, padding
                =1)
94          self.activ1 = Triangle(power=0.7)
95
96          self.res1 = HebbianResidualBlock(96, 384, kernel_size=3,
                expansion_factor=4, hebb_params=hebb_params, t_invert
                =0.65, act=1.4)
97          self.pool2 = nn.MaxPool2d(kernel_size=4, stride=2, padding
                =1)
98
99          self.res2 = HebbianResidualBlock(384, 1536, kernel_size=3,
                expansion_factor=4, hebb_params=hebb_params, t_invert
                =0.25)
100         self.pool3 = nn.AvgPool2d(kernel_size=2, stride=2, padding
                =0)
101
102         self.flatten = nn.Flatten()
103         self.fc1 = nn.Linear(24576, 10)  # Adjust this based on your
                input size
104         self.fc1.weight.data = 0.11048543456039805 * torch.rand(10,
                24576)
105         self.dropout = nn.Dropout(0.5)
106
107     def forward_features(self, x):
108         x = self.pool1(self.activ1(self.conv1(self.bn1(x))))
109         return x
110
111     def features_extract(self, x):
112         x = self.forward_features(x)
113         x = self.pool2(self.res1(x))
114         x = self.pool3(self.res2(x))
115         return x
116
117     def forward(self, x):
118         x = self.features_extract(x)
119         x = self.flatten(x)
120         x = self.fc1(self.dropout(x))
121         return x
122
123     # Plot neurons/filter of a target layer
124     def plot_grid(self, tensor, path, num_rows=5, num_cols=5,
            layer_name=""):
```

```
        # Ensure we're working with the first 25 filters (or less if there are fewer)
        excitatory = tensor[:20]
        inhibitory = tensor[-5:]
        # Symmetric normalization for excitatory weights
        max_abs_exc = torch.max(torch.abs(excitatory))
        norm_exc = excitatory / (max_abs_exc + 1e-8)
        # Symmetric normalization for inhibitory weights
        max_abs_inh = torch.max(torch.abs(inhibitory))
        norm_inh = inhibitory / (max_abs_inh + 1e-8)
        tensor = torch.cat((norm_exc, norm_inh))
        # Normalize the tensor
        # Move to CPU and convert to numpy
        tensor = tensor.cpu().detach().numpy()

        if tensor.shape[2] == 1 and tensor.shape[3] == 1:  # 1x1 convolution case
            out_channels, in_channels = tensor.shape[:2]
            fig = plt.figure(figsize=(14, 10))
            # Create a gridspec for the layout
            gs = fig.add_gridspec(2, 2, width_ratios=[20, 1], height_ratios=[1, 3],
                                  left=0.1, right=0.9, bottom=0.1, top=0.9, wspace=0.05, hspace=0.2)
            ax1 = fig.add_subplot(gs[0, 0])
            ax2 = fig.add_subplot(gs[1, 0])
            cbar_ax = fig.add_subplot(gs[:, 1])
            # Bar plot for average weights per filter
            avg_weights = tensor.mean(axis=(1, 2, 3))
            norm = plt.Normalize(vmin=avg_weights.min(), vmax=avg_weights.max())
            im1 = ax1.bar(range(out_channels), avg_weights, color=plt.cm.RdYlGn(norm(avg_weights)))
            ax1.set_xlabel('Filter Index')
            ax1.set_ylabel('Average Weight')
            ax1.set_title(f'Average Weights for 1x1 Kernels in {layer_name}')
            ax1.axhline(y=0, color='black', linestyle='--', linewidth=0.5)
            # Heatmap for detailed weight distribution
            im2 = ax2.imshow(tensor.reshape(out_channels, in_channels), cmap='RdYlGn', aspect='auto', norm=norm)
            ax2.set_xlabel('Input Channel')
            ax2.set_ylabel('Output Channel (Filter)')
            ax2.set_title('Detailed Weight Distribution')
            # Add colorbar to the right of both subplots
            fig.colorbar(im2, cax=cbar_ax, label='Normalized Weight Value')

        else:
            fig, axes = plt.subplots(num_rows, num_cols, figsize=(15, 10))
            fig.suptitle(f'First 25 Filters of {layer_name}')
            for i, ax in enumerate(axes.flat):
                if i < tensor.shape[0]:
                    filter_img = tensor[i]
```

```
                    # Handle different filter shapes
                    if filter_img.shape[0] == 3:  # RGB filter (3, H, W)
                        filter_img = np.transpose(filter_img, (1, 2, 0))
                        filter_img = (filter_img - filter_img.min()) / (filter_img.max() - filter_img.min() + 1e-8)
                    elif filter_img.shape[0] == 1:  # Grayscale filter (1, H, W)
                        filter_img = filter_img.squeeze()
                    else:  # Multi-channel filter (C, H, W), take mean across channels
                        filter_img = np.mean(filter_img, axis=0)
                    ax.imshow(filter_img, cmap='viridis' if filter_img.ndim == 2 else None)
                    ax.set_title(f'Filter {i + 1}')
                ax.axis('off')

        if path:
            fig.savefig(path, bbox_inches='tight')
        wandb.log({f'{layer_name} filters': wandb.Image(fig)})
        plt.close(fig)

        def visualize_filters(self, layer_name='conv1', save_path=None):
            weights = getattr(self, layer_name).weight.data
            self.plot_grid(weights, save_path, layer_name=layer_name)
```

A.2.9 Model_BackProp.py

```
import torch
import torch.nn as nn
import torch.nn.functional as F
from sklearn.decomposition import PCA

import matplotlib.pyplot as plt
import numpy as np
from sklearn.manifold import TSNE
from sklearn.preprocessing import StandardScaler
from tqdm import tqdm
from itertools import islice
import umap
import wandb
import math
import seaborn as sns

"""
Backpropagation model similar to model_hebb.py
TODO: make code more similar to model_hebb.py, with different architectures controlled by an argument
"""

torch.manual_seed(0)

class Triangle(nn.Module):
    def __init__(self, power: float = 1, inplace: bool = True):
```

```
26          super(Triangle, self).__init__()
27          self.inplace = inplace
28          self.power = power
29
30      def forward(self, input: torch.Tensor) -> torch.Tensor:
31          input = input - torch.mean(input.data, axis=1, keepdims=True
                )
32          return F.relu(input, inplace=self.inplace) ** self.power
33
34  def custom_depthwise_conv_init(conv_layer):
35      in_channels = conv_layer.in_channels
36      kernel_size = conv_layer.kernel_size[0]  # Assuming square
            kernels
37      weight_range = 25 / math.sqrt(in_channels * kernel_size *
            kernel_size)
38      # Depthwise separable weights
39      conv_layer.weight.data = weight_range * torch.randn(in_channels,
            1, *conv_layer.kernel_size)
40      if conv_layer.bias is not None:
41          conv_layer.bias.data.zero_()
42
43  def custom_pointwise_conv_init(conv_layer):
44      in_channels = conv_layer.in_channels
45      out_channels = conv_layer.out_channels
46      kernel_size = conv_layer.kernel_size[0]  # Should be 1 for
            pointwise
47      groups = conv_layer.groups
48      weight_range = 25 / math.sqrt(in_channels * kernel_size *
            kernel_size)
49      conv_layer.weight.data = weight_range * torch.randn(out_channels
            , in_channels // groups, *conv_layer.kernel_size)
50      if conv_layer.bias is not None:
51          conv_layer.bias.data.zero_()
52
53  class Net_Backpropagation_depth(nn.Module):
54      def __init__(self):
55          super(Net_Backpropagation_depth, self).__init__()
56
57          # A single Depthwise convolutional layer
58          self.bn1 = nn.BatchNorm2d(3, affine=False)
59          self.conv1 = nn.Conv2d(in_channels=3, out_channels=96,
                kernel_size=(5,5), padding=2, bias=False)
60          self.pool1 = nn.MaxPool2d(kernel_size=4, stride=2, padding
                =1)
61          self.activ1 = nn.ReLU()
62
63          self.bn2 = nn.BatchNorm2d(96, affine=False)
64          self.conv2 = nn.Conv2d(in_channels=96, out_channels=96,
                kernel_size=(3,3), padding=1, bias=False, groups=96)
65          self.bn_point2 = nn.BatchNorm2d(96, affine=False)
66          self.conv_point2 = nn.Conv2d(in_channels=96, out_channels
                =384, kernel_size=(1,1), padding=0, bias=False)
67          self.pool2 = nn.MaxPool2d(kernel_size=4, stride=2, padding
                =1)
68          self.activ2 = nn.ReLU()
69
70          self.bn3 = nn.BatchNorm2d(384, affine=False)
```

```
71          self.conv3 = nn.Conv2d(in_channels=384, out_channels=384,
                kernel_size=(3,3), padding=1, bias=False, groups=384)
72          self.bn_point3 = nn.BatchNorm2d(384, affine=False)
73          self.conv_point3 = nn.Conv2d(in_channels=384, out_channels
                =1536, kernel_size=(1,1), padding=0, bias=False)
74          self.pool3 = nn.AvgPool2d(kernel_size=2, stride=2, padding
                =0)
75          self.activ3 = nn.ReLU()
76
77          self.flatten = nn.Flatten()
78          # Final fully-connected layer classifier
79          self.fc1 = nn.Linear(24576, 10)
80          self.fc1.weight.data = 0.11048543456039805 * torch.rand(10,
                24576)
81          self.dropout = nn.Dropout(0.5)
82
83          # Apply custom initialization to depthwise convolutional
                layers
84          custom_pointwise_conv_init(self.conv1)
85          custom_depthwise_conv_init(self.conv2)
86          custom_depthwise_conv_init(self.conv3)
87
88          # Apply custom initialization to pointwise convolutional
                layers
89          custom_pointwise_conv_init(self.conv_point2)
90          custom_pointwise_conv_init(self.conv_point3)
91
92      def forward_features(self, x):
93          x = self.pool1(self.activ1(self.conv1(self.bn1(x))))
94          return x
95
96      def features_extract(self, x):
97          x = self.forward_features(x)
98          x = self.pool2(self.activ2(self.conv_point2(self.bn_point2(
                self.conv2(self.bn2(x))))))
99          x = self.pool3(self.activ3(self.conv_point3(self.bn_point3(
                self.conv3(self.bn3(x))))))
100         return x
101
102     def forward(self, x):
103         x = self.features_extract(x)
104         x = self.flatten(x)
105         x = self.fc1(self.dropout(x))
106         return x
107
108     def plot_grid(self, tensor, path, num_rows=5, num_cols=5,
            layer_name=""):
109         # Ensure we're working with the first 25 filters (or less if
                there are fewer)
110         tensor = tensor[:25]
111         # Normalize the tensor
112         tensor = (tensor - tensor.min()) / (tensor.max() - tensor.
                min() + 1e-8)
113         # Move to CPU and convert to numpy
114         tensor = tensor.cpu().detach().numpy()
115
116         if tensor.shape[2] == 1 and tensor.shape[3] == 1:  # 1x1
                convolution case
```

```
            out_channels, in_channels = tensor.shape[:2]
            fig = plt.figure(figsize=(14, 10))

            # Create a gridspec for the layout
            gs = fig.add_gridspec(2, 2, width_ratios=[20, 1],
                height_ratios=[1, 3],
                                  left=0.1, right=0.9, bottom=0.1,
                                      top=0.9, wspace=0.05, hspace
                                      =0.2)

            ax1 = fig.add_subplot(gs[0, 0])
            ax2 = fig.add_subplot(gs[1, 0])
            cbar_ax = fig.add_subplot(gs[:, 1])

            # Bar plot for average weights per filter
            avg_weights = tensor.mean(axis=(1, 2, 3))
            norm = plt.Normalize(vmin=avg_weights.min(), vmax=
                avg_weights.max())
            im1 = ax1.bar(range(out_channels), avg_weights, color=
                plt.cm.RdYlGn(norm(avg_weights)))
            ax1.set_xlabel('Filter Index')
            ax1.set_ylabel('Average Weight')
            ax1.set_title(f'Average Weights for 1x1 Kernels in {
                layer_name}')
            ax1.axhline(y=0, color='black', linestyle='--',
                linewidth=0.5)

            # Heatmap for detailed weight distribution
            im2 = ax2.imshow(tensor.reshape(out_channels,
                in_channels), cmap='RdYlGn', aspect='auto', norm=norm
                )
            ax2.set_xlabel('Input Channel')
            ax2.set_ylabel('Output Channel (Filter)')
            ax2.set_title('Detailed Weight Distribution')

            # Add colorbar to the right of both subplots
            fig.colorbar(im2, cax=cbar_ax, label='Normalized Weight
                Value')

        else:
            fig, axes = plt.subplots(num_rows, num_cols, figsize
                =(15, 10))
            fig.suptitle(f'First 25 Filters of {layer_name}')
            for i, ax in enumerate(axes.flat):
                if i < tensor.shape[0]:
                    filter_img = tensor[i]
                    # Handle different filter shapes
                    if filter_img.shape[0] == 3:  # RGB filter (3, H
                        , W)
                        filter_img = np.transpose(filter_img, (1, 2,
                            0))
                    elif filter_img.shape[0] == 1:  # Grayscale
                        filter (1, H, W)
                        filter_img = filter_img.squeeze()
                    else:  # Multi-channel filter (C, H, W), take
                        mean across channels
                        filter_img = np.mean(filter_img, axis=0)
```

```
                    ax.imshow(filter_img, cmap='viridis' if
                        filter_img.ndim == 2 else None)
                    ax.set_title(f'Filter {i + 1}')
                ax.axis('off')

        if path:
            fig.savefig(path, bbox_inches='tight')
        wandb.log({f'{layer_name} filters': wandb.Image(fig)})
        plt.close(fig)

    def visualize_filters(self, layer_name='conv1', save_path=None):
        weights = getattr(self, layer_name).weight.data
        self.plot_grid(weights, save_path, layer_name=layer_name)


class Net_Backpropagation(nn.Module):
    def __init__(self):
        super(Net_Backpropagation, self).__init__()

        # A single Depthwise convolutional layer
        self.bn1 = nn.BatchNorm2d(3, affine=False)
        self.conv1 = nn.Conv2d(in_channels=3, out_channels=96,
            kernel_size=(5,5), padding=2, bias=False)
        self.pool1 = nn.MaxPool2d(kernel_size=4, stride=2, padding
            =1)
        self.activ1 = nn.ReLU()

        self.bn2 = nn.BatchNorm2d(96, affine=False)
        self.conv2 = nn.Conv2d(in_channels=96, out_channels=384,
            kernel_size=(3,3), padding=1, bias=False)
        self.pool2 = nn.MaxPool2d(kernel_size=4, stride=2, padding
            =1)
        self.activ2 = nn.ReLU()

        self.bn3 = nn.BatchNorm2d(384, affine=False)
        self.conv3 = nn.Conv2d(in_channels=384, out_channels=1536,
            kernel_size=(3,3), padding=1, bias=False)
        self.pool3 = nn.AvgPool2d(kernel_size=2, stride=2, padding
            =0)
        self.activ3 = nn.ReLU()

        self.flatten = nn.Flatten()
        # Final fully-connected layer classifier
        self.fc1 = nn.Linear(24576, 10)
        self.fc1.weight.data = 0.11048543456039805 * torch.rand(10,
            24576)
        self.dropout = nn.Dropout(0.5)

        # Apply custom initialization to depthwise convolutional
            layers
        custom_pointwise_conv_init(self.conv1)
        custom_pointwise_conv_init(self.conv2)
        custom_pointwise_conv_init(self.conv3)

    def forward_features(self, x):
        x = self.pool1(self.activ1(self.conv1(self.bn1(x))))
        return x

```

```
208     def features_extract(self, x):
209         x = self.forward_features(x)
210         x = self.pool2(self.activ2(self.conv2(self.bn2(x))))
211         x = self.pool3(self.activ3(self.conv3(self.bn3(x))))
212         return x
213 
214     def forward(self, x):
215         x = self.features_extract(x)
216         x = self.flatten(x)
217         x = self.fc1(self.dropout(x))
218         return x
219 
220     def plot_grid(self, tensor, path, num_rows=5, num_cols=5,
        layer_name=""):
221         # Ensure we're working with the first 25 filters (or less if
                there are fewer)
222         tensor = tensor[:25]
223         # Normalize the tensor
224         tensor = (tensor - tensor.min()) / (tensor.max() - tensor.
            min() + 1e-8)
225         # Move to CPU and convert to numpy
226         tensor = tensor.cpu().detach().numpy()
227 
228         if tensor.shape[2] == 1 and tensor.shape[3] == 1:  # 1x1
            convolution case
229             out_channels, in_channels = tensor.shape[:2]
230             fig = plt.figure(figsize=(14, 10))
231 
232             # Create a gridspec for the layout
233             gs = fig.add_gridspec(2, 2, width_ratios=[20, 1],
                height_ratios=[1, 3],
234                                   left=0.1, right=0.9, bottom=0.1,
                                      top=0.9, wspace=0.05, hspace
                                      =0.2)
235 
236             ax1 = fig.add_subplot(gs[0, 0])
237             ax2 = fig.add_subplot(gs[1, 0])
238             cbar_ax = fig.add_subplot(gs[:, 1])
239 
240             # Bar plot for average weights per filter
241             avg_weights = tensor.mean(axis=(1, 2, 3))
242             norm = plt.Normalize(vmin=avg_weights.min(), vmax=
                avg_weights.max())
243             im1 = ax1.bar(range(out_channels), avg_weights, color=
                plt.cm.RdYlGn(norm(avg_weights)))
244             ax1.set_xlabel('Filter Index')
245             ax1.set_ylabel('Average Weight')
246             ax1.set_title(f'Average Weights for 1x1 Kernels in {
                layer_name}')
247             ax1.axhline(y=0, color='black', linestyle='--',
                linewidth=0.5)
248 
249             # Heatmap for detailed weight distribution
250             im2 = ax2.imshow(tensor.reshape(out_channels,
                in_channels), cmap='RdYlGn', aspect='auto', norm=norm
                )
251             ax2.set_xlabel('Input Channel')
252             ax2.set_ylabel('Output Channel (Filter)')
```

```
            ax2.set_title('Detailed Weight Distribution')

            # Add colorbar to the right of both subplots
            fig.colorbar(im2, cax=cbar_ax, label='Normalized Weight Value')

        else:
            fig, axes = plt.subplots(num_rows, num_cols, figsize=(15, 10))
            fig.suptitle(f'First 25 Filters of {layer_name}')
            for i, ax in enumerate(axes.flat):
                if i < tensor.shape[0]:
                    filter_img = tensor[i]
                    # Handle different filter shapes
                    if filter_img.shape[0] == 3:  # RGB filter (3, H, W)
                        filter_img = np.transpose(filter_img, (1, 2, 0))
                    elif filter_img.shape[0] == 1:  # Grayscale filter (1, H, W)
                        filter_img = filter_img.squeeze()
                    else:  # Multi-channel filter (C, H, W), take mean across channels
                        filter_img = np.mean(filter_img, axis=0)
                    ax.imshow(filter_img, cmap='viridis' if filter_img.ndim == 2 else None)
                    ax.set_title(f'Filter {i + 1}')
                ax.axis('off')

        if path:
            fig.savefig(path, bbox_inches='tight')
        wandb.log({f'{layer_name} filters': wandb.Image(fig)})
        plt.close(fig)

    def visualize_filters(self, layer_name='conv1', save_path=None):
        weights = getattr(self, layer_name).weight.data
        self.plot_grid(weights, save_path, layer_name=layer_name)
```

A.2.10 Experiment_hebbian.py

```
import torch
import torch.nn as nn
import torch.optim as optim
import torch.optim.lr_scheduler as sched

import data
from model_hebb import Net_Hebbian
import matplotlib.pyplot as plt
import warnings

from logger import Logger
from torchmetrics import Accuracy, Precision, Recall, F1Score, ConfusionMatrix
import seaborn as sns
import wandb
from visualizer import plot_ltp_ltd, print_weight_statistics, visualize_data_clusters
from receptive_fields import visualize_filters
```

```
17
18 """
19 Code structure inspired and modified from https://github.com/
       NeuromorphicComputing/SoftHebb
20 Similar training and testing loop, with some identical function for
       the custom learning rate schedule
21 """
22
23 torch.manual_seed(0)
24
25 # Calculate evaluation metrics
26 def calculate_metrics(preds, labels, num_classes):
27     if num_classes == 2:
28         accuracy = Accuracy(task='binary', num_classes=num_classes).
              to(device)
29         precision = Precision(task='binary', average='weighted',
              num_classes=num_classes).to(device)
30         recall = Recall(task='binary', average='weighted',
              num_classes=num_classes).to(device)
31         f1 = F1Score(task='binary', average='weighted', num_classes=
              num_classes).to(device)
32         confusion_matrix = ConfusionMatrix(task='binary',
              num_classes=num_classes).to(device)
33     else:
34         accuracy = Accuracy(task='multiclass', num_classes=
              num_classes).to(device)
35         precision = Precision(task='multiclass', average='macro',
              num_classes=num_classes).to(device)
36         recall = Recall(task='multiclass', average='macro',
              num_classes=num_classes).to(device)
37         f1 = F1Score(task='multiclass', average='macro', num_classes
              =num_classes).to(device)
38         confusion_matrix = ConfusionMatrix(task='multiclass',
              num_classes=num_classes).to(device)
39
40     acc = accuracy(preds, labels)
41     prec = precision(preds, labels)
42     rec = recall(preds, labels)
43     f1_score = f1(preds, labels)
44     conf_matrix = confusion_matrix(preds, labels)
45
46     return acc, prec, rec, f1_score, conf_matrix
47
48 class WeightNormDependentLR(optim.lr_scheduler._LRScheduler):
49     """
50     Custom Learning Rate Scheduler for unsupervised training of
          SoftHebb Convolutional blocks.
51     Difference between current neuron norm and theoretical converged
           norm (=1) scales the initial lr.
52     """
53
54     def __init__(self, optimizer, power_lr, last_epoch=-1, verbose=
          False):
55         self.optimizer = optimizer
56         self.initial_lr_groups = [group['lr'] for group in self.
              optimizer.param_groups]  # store initial lrs
57         self.power_lr = power_lr
58         super().__init__(optimizer, last_epoch, verbose)
```

```
59
60      def get_lr(self):
61          if not self._get_lr_called_within_step:
62              warnings.warn("To get the last learning rate computed by
                    the scheduler, "
63                            "please use `get_last_lr()`.", UserWarning
                                )
64          new_lr = []
65          for i, group in enumerate(self.optimizer.param_groups):
66              for param in group['params']:
67                  # difference between current neuron norm and
                        theoretical converged norm (=1) scales the
                        initial lr
68                  # initial_lr * |neuron_norm - 1| ** 0.5
69                  norm_diff = torch.abs(torch.linalg.norm(param.view(
                        param.shape[0], -1), dim=1, ord=2) - 1) + 1e-10
70                  new_lr.append(self.initial_lr_groups[i] * (norm_diff
                         ** self.power_lr)[:, None, None, None])
71          return new_lr
72
73  class TensorLRSGD(optim.SGD):
74      @torch.no_grad()
75      def step(self, closure=None):
76          """Performs a single optimization step, using a non-scalar (
                tensor) learning rate.
77          Arguments:
78              closure (callable, optional): A closure that reevaluates
                    the model
79                  and returns the loss.
80          """
81          loss = None
82          if closure is not None:
83              with torch.enable_grad():
84                  loss = closure()
85          for group in self.param_groups:
86              weight_decay = group['weight_decay']
87              momentum = group['momentum']
88              dampening = group['dampening']
89              nesterov = group['nesterov']
90              for p in group['params']:
91                  if p.grad is None:
92                      continue
93                  d_p = p.grad
94                  if weight_decay != 0:
95                      d_p = d_p.add(p, alpha=weight_decay)
96                  if momentum != 0:
97                      param_state = self.state[p]
98                      if 'momentum_buffer' not in param_state:
99                          buf = param_state['momentum_buffer'] = torch
                                .clone(d_p).detach()
100                     else:
101                         buf = param_state['momentum_buffer']
102                         buf.mul_(momentum).add_(d_p, alpha=1 -
                                dampening)
103                     if nesterov:
104                         d_p = d_p.add(buf, alpha=momentum)
105                     else:
106                         d_p = buf
```

```
                p.add_(-group['lr'] * d_p)
        return loss

# Main training and testing loop
# Experiment is configured to replicate SoftHebb-Surr/HardWTA/Cos-Instar configuration results
if __name__ == "__main__":

    hebb_param = {'mode': 'hard', 'w_nrm': False, 'act': nn.Identity(), 'k': 1, 'alpha': 1.}
    device = torch.device('cuda:0')
    model = Net_Hebbian(hebb_params=hebb_param, version="softhebb")
    model.to(device)

    wandb_logger = Logger(
        f"SoftHebb-Surr/HardWTA/Cos-Instar",project='Final-ReRuns-HebbianCNN', model=model)
    logger = wandb_logger.get_logger()
    num_parameters = sum(p.numel() for p in model.parameters() if p.requires_grad)
    print(f"Parameter Count Total: {num_parameters}")

    # Custom learning rate and scheduler for only SoftHebb implementation
    # unsup_optimizer = TensorLRSGD([
    #     {"params": model.conv1.parameters(), "lr": 0.08, },
    #     {"params": model.conv2.parameters(), "lr": 0.005, },
    #     {"params": model.conv3.parameters(), "lr": 0.01, },
    # ], lr=0)
    # unsup_lr_scheduler = WeightNormDependentLR(unsup_optimizer, power_lr=0.5)

    # Learning rate for all other hebbian experiments
    hebb_params = [
        {'params': model.conv1.parameters(), 'lr': 0.1},
        {'params': model.conv2.parameters(), 'lr': 0.1},
        {'params': model.conv3.parameters(), 'lr': 0.1}
    ]
    unsup_optimizer = optim.SGD(hebb_params, lr=0)  # The lr here will be overridden by the individual lrs

    sup_optimizer = optim.Adam(model.fc1.parameters(), lr=0.001)
    criterion = nn.CrossEntropyLoss()

    trn_set, tst_set, zca = data.get_data(dataset='cifar10', root='datasets', batch_size=64,
                                          whiten_lvl=1e-3)
    print(f'Processing Training batches: {len(trn_set)}')

    print("Initial Weight statistics")
    print_weight_statistics(model.conv1, 'conv1')
    print_weight_statistics(model.conv2, 'conv2')
    print_weight_statistics(model.conv3, 'conv3')

    print("Visualizing Initial Filters")
    model.visualize_filters('conv1')

    running_loss = 0.0
```

```
    # Hebbian feature extraction
    for epoch in range(1):
        print(f"Training Hebbian epoch {epoch}")
        for i, data in enumerate(trn_set, 0):
            inputs, _ = data
            inputs = inputs.to(device)
            # zero the parameter gradients
            unsup_optimizer.zero_grad()
            with torch.no_grad():
                outputs = model(inputs)
            # Visualize changes before updating
            if i % 200 == 0: # Every 100 datapoint
                print(f'Saving details after batch {i}')
                plot_ltp_ltd(model.conv1, 'conv1', num_filters=10,
                    detailed_mode=True)
                plot_ltp_ltd(model.conv2, 'conv2', num_filters=10,
                    detailed_mode=True)
                plot_ltp_ltd(model.conv3, 'conv3', num_filters=10,
                    detailed_mode=True)

                model.visualize_filters('conv1')
                model.visualize_filters('conv2')
                model.visualize_filters('conv3')
            for layer in [model.conv1, model.conv2, model.conv3]:
                if hasattr(layer, 'local_update'):
                    layer.local_update()
            unsup_optimizer.step()
            # Scheduler only for SoftHebb
            # unsup_lr_scheduler.step()
    print("Visualizing Filters")
    model.visualize_filters('conv1', f'results/{"demo"}/
        demo_conv1_filters_epoch_{1}.png')
    model.visualize_filters('conv2', f'results/{"demo"}/
        demo_conv2_filters_epoch_{1}.png')
    model.visualize_filters('conv3', f'results/{"demo"}/
        demo_conv3_filters_epoch_{1}.png')

    # set requires grad false and eval mode for all modules but
        classifier
    unsup_optimizer.zero_grad()
    model.conv1.requires_grad = False
    model.conv2.requires_grad = False
    model.conv3.requires_grad = False
    # model.conv4.requires_grad = False
    model.conv1.eval()
    model.conv2.eval()
    model.conv3.eval()
    # model.conv4.eval()
    model.bn1.eval()
    model.bn2.eval()
    model.bn3.eval()
    # model.bn4.eval()
    print("Visualizing Test Class separation")
    visualize_data_clusters(tst_set, model=model, method='umap', dim
        =2)
    # Supervised training of classifier
    # Train classifier with backpropagation
    print("Training Classifier")
```

```
    for epoch in range(10):
        model.fc1.train()
        model.dropout.train()
        running_loss = 0.0
        correct = 0
        total = 0
        train_preds = []
        train_labels = []
        for i, data in enumerate(trn_set, 0):
            inputs, labels = data
            inputs = inputs.to(device)
            labels = labels.to(device)
            # zero the parameter gradients
            sup_optimizer.zero_grad()
            outputs = model(inputs)
            loss = criterion(outputs, labels)
            loss.backward()
            sup_optimizer.step()
            # compute training statistics
            running_loss += loss.item()
            total += labels.size(0)
            _, predicted = torch.max(outputs.data, 1)
            correct += (predicted == labels).sum().item()
            # For wandb logs
            preds = torch.argmax(outputs, dim=1)
            train_preds.append(preds)
            train_labels.append(labels)

        print(f'Accuracy of the network on the train images: {100 *
            correct // total} %')
        print(f'[{epoch + 1}] loss: {running_loss / total:.3f}')

        train_preds = torch.cat(train_preds, dim=0)
        train_labels = torch.cat(train_labels, dim=0)
        acc, prec, rec, f1_score, conf_matrix = calculate_metrics(
            train_preds, train_labels, 10)
        logger.log({'train_accuracy': acc, 'train_precision': prec,
            'train_recall': rec, 'train_f1_score': f1_score})
        f, ax = plt.subplots(figsize=(15, 10))
        sns.heatmap(conf_matrix.clone().detach().cpu().numpy(),
            annot=True, ax=ax)
        logger.log({"train_confusion_matrix": wandb.Image(f)})
        plt.close(f)

        # Evaluation on test set
        model.eval()
        running_loss = 0.
        correct = 0
        total = 0
        test_preds = []
        test_labels = []
        # since we're not training, we don't need to calculate the
            gradients for our outputs/model
        with torch.no_grad():
            for data in tst_set:
                images, labels = data
                images = images.to(device)
                labels = labels.to(device)
```

```
                # calculate outputs by running images through the
                    network
                outputs = model(images)
                # the class with the highest energy is what we
                    choose as prediction
                _, predicted = torch.max(outputs.data, 1)
                total += labels.size(0)
                correct += (predicted == labels).sum().item()
                loss = criterion(outputs, labels)
                running_loss += loss.item()
                # For wandb logs
                preds = torch.argmax(outputs, dim=1)
                test_preds.append(preds)
                test_labels.append(labels)

        print(f'Accuracy of the network on the test images: {100 *
            correct / total} %')
        print(f'test loss: {running_loss / total:.3f}')

        test_preds = torch.cat(test_preds, dim=0)
        test_labels = torch.cat(test_labels, dim=0)
        acc, prec, rec, f1_score, conf_matrix = calculate_metrics(
            test_preds, test_labels, 10)
        logger.log({'test_accuracy': acc, 'test_precision': prec, '
            test_recall': rec, 'test_f1_score': f1_score})
        f, ax = plt.subplots(figsize=(15, 10))
        sns.heatmap(conf_matrix.clone().detach().cpu().numpy(),
            annot=True, ax=ax)
        logger.log({"test_confusion_matrix": wandb.Image(f)})
        plt.close(f)

    # Visualise receptive fields at end, as this code alters the
        model architecture (padding removal)
    print("Visualizing Receptive fields")
    visualize_filters(model, model.conv1, num_filters=25)
    visualize_filters(model, model.conv2, num_filters=25)
    visualize_filters(model, model.conv3, num_filters=25)
```

A.2.11 Experiment_hebbian_depthwise.py

```
import torch
import torch.nn as nn
import torch.optim as optim
import torch.optim.lr_scheduler as sched

import data
from model_depthwise import Net_Depthwise
import numpy as np
import matplotlib.pyplot as plt
import warnings

from logger import Logger
from torchmetrics import Accuracy, Precision, Recall, F1Score,
    ConfusionMatrix
import seaborn as sns
import wandb
from visualizer import plot_ltp_ltd, plot_ltp_ltd_ex_in,
    print_weight_statistics, visualize_data_clusters
```

```
import pandas as pd
from receptive_fields import visualize_filters

"""
Identical to experimen_hebbian.py, but adapted for depthwise convolutions
"""

torch.manual_seed(0)

def calculate_metrics(preds, labels, num_classes):
    if num_classes == 2:
        accuracy = Accuracy(task='binary', num_classes=num_classes).to(device)
        precision = Precision(task='binary', average='weighted', num_classes=num_classes).to(device)
        recall = Recall(task='binary', average='weighted', num_classes=num_classes).to(device)
        f1 = F1Score(task='binary', average='weighted', num_classes=num_classes).to(device)
        confusion_matrix = ConfusionMatrix(task='binary', num_classes=num_classes).to(device)
    else:
        accuracy = Accuracy(task='multiclass', num_classes=num_classes).to(device)
        precision = Precision(task='multiclass', average='macro', num_classes=num_classes).to(device)
        recall = Recall(task='multiclass', average='macro', num_classes=num_classes).to(device)
        f1 = F1Score(task='multiclass', average='macro', num_classes=num_classes).to(device)
        confusion_matrix = ConfusionMatrix(task='multiclass', num_classes=num_classes).to(device)

    acc = accuracy(preds, labels)
    prec = precision(preds, labels)
    rec = recall(preds, labels)
    f1_score = f1(preds, labels)
    conf_matrix = confusion_matrix(preds, labels)

    return acc, prec, rec, f1_score, conf_matrix

class WeightNormDependentLR(optim.lr_scheduler._LRScheduler):
    """
    Custom Learning Rate Scheduler for unsupervised training of SoftHebb Convolutional blocks.
    Difference between current neuron norm and theoretical converged norm (=1) scales the initial lr.
    """

    def __init__(self, optimizer, power_lr, last_epoch=-1, verbose=False):
        self.optimizer = optimizer
        self.initial_lr_groups = [group['lr'] for group in self.optimizer.param_groups]  # store initial lrs
        self.power_lr = power_lr
        super().__init__(optimizer, last_epoch, verbose)

```

```
60      def get_lr(self):
61          if not self._get_lr_called_within_step:
62              warnings.warn("To get the last learning rate computed by
                    the scheduler, "
63                            "please use `get_last_lr()`.", UserWarning
                               )
64          new_lr = []
65          for i, group in enumerate(self.optimizer.param_groups):
66              for param in group['params']:
67                  # difference between current neuron norm and
                        theoretical converged norm (=1) scales the
                        initial lr
68                  # initial_lr * |neuron_norm - 1| ** 0.5
69                  norm_diff = torch.abs(torch.linalg.norm(param.view(
                        param.shape[0], -1), dim=1, ord=2) - 1) + 1e-10
70                  new_lr.append(self.initial_lr_groups[i] * (norm_diff
                         ** self.power_lr)[:, None, None, None])
71          return new_lr
72
73
74  class TensorLRSGD(optim.SGD):
75      @torch.no_grad()
76      def step(self, closure=None):
77          """Performs a single optimization step, using a non-scalar (
                tensor) learning rate.
78          Arguments:
79              closure (callable, optional): A closure that reevaluates
                    the model
80                  and returns the loss.
81          """
82          loss = None
83          if closure is not None:
84              with torch.enable_grad():
85                  loss = closure()
86          for group in self.param_groups:
87              weight_decay = group['weight_decay']
88              momentum = group['momentum']
89              dampening = group['dampening']
90              nesterov = group['nesterov']
91              for p in group['params']:
92                  if p.grad is None:
93                      continue
94                  d_p = p.grad
95                  if weight_decay != 0:
96                      d_p = d_p.add(p, alpha=weight_decay)
97                  if momentum != 0:
98                      param_state = self.state[p]
99                      if 'momentum_buffer' not in param_state:
100                         buf = param_state['momentum_buffer'] = torch
                                .clone(d_p).detach()
101                     else:
102                         buf = param_state['momentum_buffer']
103                         buf.mul_(momentum).add_(d_p, alpha=1 -
                                dampening)
104                     if nesterov:
105                         d_p = d_p.add(buf, alpha=momentum)
106                     else:
107                         d_p = buf
```

```
                p.add_(-group['lr'] * d_p)
        return loss

if __name__ == "__main__":

    hebb_param = {'mode': 'hard', 'w_nrm': False, 'act': nn.Identity
        (), 'k': 1, 'alpha': 1.}
    device = torch.device('cuda:0')
    model = Net_Depthwise(hebb_params=hebb_param, version="softhebb"
        )
    model.to(device)

    wandb_logger = Logger(
        f"Depthwise_HardHebb-Surr/HardWTA/Cos-Instar",project='Final
            -ReRuns-HebbianCNN', model=model)
    logger = wandb_logger.get_logger()
    num_parameters = sum(p.numel() for p in model.parameters() if p.
        requires_grad)
    print(f"Parameter Count Total: {num_parameters}")

    # unsup_optimizer = TensorLRSGD([
    #     {"params": model.conv1.parameters(), "lr": 0.08, },
    #     {"params": model.conv2.parameters(), "lr": 0.005, },
    #     {"params": model.conv_point2.parameters(), "lr": 0.005, },
    #     {"params": model.conv3.parameters(), "lr": 0.01, },
    #     {"params": model.conv_point3.parameters(), "lr": 0.01, }
    # ], lr=0)
    # unsup_lr_scheduler = WeightNormDependentLR(unsup_optimizer,
        power_lr=0.5)

    hebb_params = [
        {'params': model.conv1.parameters(), 'lr': 0.1},
        {'params': model.conv2.parameters(), 'lr': 0.1},
        {'params': model.conv_point2.parameters(), 'lr': 0.1},
        {'params': model.conv3.parameters(), 'lr': 0.1},
        {'params': model.conv_point3.parameters(), 'lr': 0.1}
    ]
    unsup_optimizer = optim.SGD(hebb_params, lr=0)  # The lr here
        will be overridden by the individual lrs

    sup_optimizer = optim.Adam(model.fc1.parameters(), lr=0.001)
    criterion = nn.CrossEntropyLoss()

    trn_set, tst_set, zca = data.get_data(dataset='cifar10', root='
        datasets', batch_size=64,
                                          whiten_lvl=1e-3)
    print(f'Processing Training batches: {len(trn_set)}')

    print("Initial Weight statistics")
    print_weight_statistics(model.conv1, 'conv1')
    print_weight_statistics(model.conv2, 'conv2')
    print_weight_statistics(model.conv3, 'conv3')
    print_weight_statistics(model.conv_point2, 'conv_point2')

    running_loss = 0.0
    for epoch in range(1):
        print(f"Training Hebbian epoch {epoch}")
        for i, data in enumerate(trn_set, 0):
```

```
159             inputs, _ = data
160             inputs = inputs.to(device)
161             # zero the parameter gradients
162             # unsup_optimizer.zero_grad()
163             with torch.no_grad():
164                 outputs = model(inputs)
165             # Visualize changes before updating
166             if i % 200 == 0: # Every 100 datapoint
167                 print(f'Saving details after batch {i}')
168                 plot_ltp_ltd(model.conv1, 'conv1', num_filters=10,
                        detailed_mode=True)
169                 plot_ltp_ltd(model.conv2, 'conv2', num_filters=10,
                        detailed_mode=True)
170                 # plot_ltp_ltd(model.conv_point2, 'conv_point2',
                        num_filters=10, detailed_mode=True)
171                 model.visualize_filters('conv1')
172                 model.visualize_filters('conv2')
173                 model.visualize_filters('conv_point2')
174 
175                 model.visualize_filters('conv3')
176                 model.visualize_filters('conv_point3')
177             for layer in [model.conv1, model.conv2, model.conv3,
                    model.conv_point2, model.conv_point3]:
178             # for layer in [model.conv1, model.conv2, model.conv3]:
179                 if hasattr(layer, 'local_update'):
180                     layer.local_update()
181             # unsup_optimizer.step()
182             # unsup_lr_scheduler.step()
183     print("Visualizing Filters")
184     model.visualize_filters('conv1', f'results/{"demo"}/
            demo_conv1_filters_epoch_{1}.png')
185     model.visualize_filters('conv2', f'results/{"demo"}/
            demo_conv2_filters_epoch_{1}.png')
186     model.visualize_filters('conv3', f'results/{"demo"}/
            demo_conv3_filters_epoch_{1}.png')
187     # model.visualize_filters('conv_point2', f'results/{"demo"}/
            demo_conv_point2_filters_epoch_{1}.png')
188 
189 
190     # Supervised training of classifier
191     # set requires grad false and eval mode for all modules but
            classifier
192     unsup_optimizer.zero_grad()
193     model.conv1.requires_grad = False
194     model.conv2.requires_grad = False
195     model.conv3.requires_grad = False
196     # model.conv4.requires_grad = False
197     model.conv1.eval()
198     model.conv2.eval()
199     model.conv3.eval()
200     # model.conv4.eval()
201     model.bn1.eval()
202     model.bn2.eval()
203     model.bn3.eval()
204     # model.bn4.eval()
205     model.conv_point2.requires_grad = False
206     model.conv_point3.requires_grad = False
207     # model.conv_point4.requires_grad = False
```

```
208     model.conv_point2.eval()
209     model.conv_point3.eval()
210     # model.conv_point4.eval()
211     model.bn_point2.eval()
212     model.bn_point3.eval()
213     # model.bn_point4.eval()
214     print("Visualizing Test Class separation")
215     visualize_data_clusters(tst_set, model=model, method='umap', dim
            =2)
216     # Train classifier with backpropagation
217     print("Training Classifier")
218     for epoch in range(5):
219         model.fc1.train()
220         model.dropout.train()
221         running_loss = 0.0
222         correct = 0
223         total = 0
224         train_preds = []
225         train_labels = []
226         for i, data in enumerate(trn_set, 0):
227             inputs, labels = data
228             inputs = inputs.to(device)
229             labels = labels.to(device)
230             # zero the parameter gradients
231             sup_optimizer.zero_grad()
232             outputs = model(inputs)
233             loss = criterion(outputs, labels)
234             loss.backward()
235             sup_optimizer.step()
236             # compute training statistics
237             running_loss += loss.item()
238             total += labels.size(0)
239             _, predicted = torch.max(outputs.data, 1)
240             correct += (predicted == labels).sum().item()
241             # For wandb logs
242             preds = torch.argmax(outputs, dim=1)
243             train_preds.append(preds)
244             train_labels.append(labels)
245 
246         print(f'Accuracy of the network on the train images: {100 *
                correct // total} %')
247         print(f'[{epoch + 1}] loss: {running_loss / total:.3f}')
248 
249         train_preds = torch.cat(train_preds, dim=0)
250         train_labels = torch.cat(train_labels, dim=0)
251         acc, prec, rec, f1_score, conf_matrix = calculate_metrics(
                train_preds, train_labels, 10)
252         logger.log({'train_accuracy': acc, 'train_precision': prec,
                'train_recall': rec, 'train_f1_score': f1_score})
253         f, ax = plt.subplots(figsize=(15, 10))
254         sns.heatmap(conf_matrix.clone().detach().cpu().numpy(),
                annot=True, ax=ax)
255         logger.log({"train_confusion_matrix": wandb.Image(f)})
256         plt.close(f)
257 
258         # Evaluation on test set
259         model.eval()
260         running_loss = 0.
```

```
261         correct = 0
262         total = 0
263         test_preds = []
264         test_labels = []
265         # since we're not training, we don't need to calculate the
                gradients for our outputs
266         with torch.no_grad():
267             for data in tst_set:
268                 images, labels = data
269                 images = images.to(device)
270                 labels = labels.to(device)
271                 # calculate outputs by running images through the
                        network
272                 outputs = model(images)
273                 # the class with the highest energy is what we
                        choose as prediction
274                 _, predicted = torch.max(outputs.data, 1)
275                 total += labels.size(0)
276                 correct += (predicted == labels).sum().item()
277                 loss = criterion(outputs, labels)
278                 running_loss += loss.item()
279                 # For wandb logs
280                 preds = torch.argmax(outputs, dim=1)
281                 test_preds.append(preds)
282                 test_labels.append(labels)
283
284         print(f'Accuracy of the network on the test images: {100 *
                correct / total} %')
285         print(f'test loss: {running_loss / total:.3f}')
286
287         test_preds = torch.cat(test_preds, dim=0)
288         test_labels = torch.cat(test_labels, dim=0)
289         acc, prec, rec, f1_score, conf_matrix = calculate_metrics(
                test_preds, test_labels, 10)
290         logger.log({'test_accuracy': acc, 'test_precision': prec, '
                test_recall': rec, 'test_f1_score': f1_score})
291         f, ax = plt.subplots(figsize=(15, 10))
292         sns.heatmap(conf_matrix.clone().detach().cpu().numpy(),
                annot=True, ax=ax)
293         logger.log({"test_confusion_matrix": wandb.Image(f)})
294         plt.close(f)
295
296     print("Visualizing Receptive fields")
297     visualize_filters(model, model.conv1, num_filters=25)
298     visualize_filters(model, model.conv2, num_filters=25)
299     visualize_filters(model, model.conv3, num_filters=25)
300
301     visualize_filters(model, model.conv_point2, num_filters=25)
302     visualize_filters(model, model.conv_point3, num_filters=25)
```

A.2.12 Experiment_hebbian_residual.py

```
1 import torch
2 import torch.nn as nn
3 import torch.optim as optim
4 import torch.optim.lr_scheduler as sched
5
6 import data
```

```
from model_residual import Net_Depthwise_Residual
import numpy as np
import matplotlib.pyplot as plt
import warnings

from logger import Logger
from torchmetrics import Accuracy, Precision, Recall, F1Score,
    ConfusionMatrix
import seaborn as sns
import wandb
from visualizer import plot_ltp_ltd, print_weight_statistics,
    visualize_data_clusters
import pandas as pd
from receptive_fields_residual import visualize_filters

"""
Identical to experimen_hebbian.py, but adapted for depthwise
    convolutions
"""

torch.manual_seed(0)

def calculate_metrics(preds, labels, num_classes):
    if num_classes == 2:
        accuracy = Accuracy(task='binary', num_classes=num_classes).
            to(device)
        precision = Precision(task='binary', average='weighted',
            num_classes=num_classes).to(device)
        recall = Recall(task='binary', average='weighted',
            num_classes=num_classes).to(device)
        f1 = F1Score(task='binary', average='weighted', num_classes=
            num_classes).to(device)
        confusion_matrix = ConfusionMatrix(task='binary',
            num_classes=num_classes).to(device)
    else:
        accuracy = Accuracy(task='multiclass', num_classes=
            num_classes).to(device)
        precision = Precision(task='multiclass', average='macro',
            num_classes=num_classes).to(device)
        recall = Recall(task='multiclass', average='macro',
            num_classes=num_classes).to(device)
        f1 = F1Score(task='multiclass', average='macro', num_classes
            =num_classes).to(device)
        confusion_matrix = ConfusionMatrix(task='multiclass',
            num_classes=num_classes).to(device)

    acc = accuracy(preds, labels)
    prec = precision(preds, labels)
    rec = recall(preds, labels)
    f1_score = f1(preds, labels)
    conf_matrix = confusion_matrix(preds, labels)

    return acc, prec, rec, f1_score, conf_matrix


class WeightNormDependentLR(optim.lr_scheduler._LRScheduler):
    """
```

```
    Custom Learning Rate Scheduler for unsupervised training of SoftHebb Convolutional blocks.
    Difference between current neuron norm and theoretical converged norm (=1) scales the initial lr.
    """

    def __init__(self, optimizer, power_lr, last_epoch=-1, verbose=False):
        self.optimizer = optimizer
        self.initial_lr_groups = [group['lr'] for group in self.optimizer.param_groups]  # store initial lrs
        self.power_lr = power_lr
        super().__init__(optimizer, last_epoch, verbose)

    def get_lr(self):
        if not self._get_lr_called_within_step:
            warnings.warn("To get the last learning rate computed by the scheduler, "
                          "please use `get_last_lr()`.", UserWarning)
        new_lr = []
        for i, group in enumerate(self.optimizer.param_groups):
            for param in group['params']:
                # difference between current neuron norm and theoretical converged norm (=1) scales the initial lr
                # initial_lr * |neuron_norm - 1| ** 0.5
                norm_diff = torch.abs(torch.linalg.norm(param.view(param.shape[0], -1), dim=1, ord=2) - 1) + 1e-10
                new_lr.append(self.initial_lr_groups[i] * (norm_diff ** self.power_lr)[:, None, None, None])
        return new_lr


class TensorLRSGD(optim.SGD):
    @torch.no_grad()
    def step(self, closure=None):
        """Performs a single optimization step, using a non-scalar (tensor) learning rate.
        Arguments:
            closure (callable, optional): A closure that reevaluates the model
                and returns the loss.
        """
        loss = None
        if closure is not None:
            with torch.enable_grad():
                loss = closure()
        for group in self.param_groups:
            weight_decay = group['weight_decay']
            momentum = group['momentum']
            dampening = group['dampening']
            nesterov = group['nesterov']
            for p in group['params']:
                if p.grad is None:
                    continue
                d_p = p.grad
                if weight_decay != 0:
```

```
                    d_p = d_p.add(p, alpha=weight_decay)
                if momentum != 0:
                    param_state = self.state[p]
                    if 'momentum_buffer' not in param_state:
                        buf = param_state['momentum_buffer'] = torch
                            .clone(d_p).detach()
                    else:
                        buf = param_state['momentum_buffer']
                        buf.mul_(momentum).add_(d_p, alpha=1 -
                            dampening)
                    if nesterov:
                        d_p = d_p.add(buf, alpha=momentum)
                    else:
                        d_p = buf
                p.add_(-group['lr'] * d_p)
        return loss


if __name__ == "__main__":

    hebb_param = {'mode': 'hard', 'w_nrm': False, 'act': nn.Identity
        (), 'k': 1, 'alpha': 1.}
    device = torch.device('cuda:0')
    model = Net_Depthwise_Residual(hebb_params=hebb_param)
    model.to(device)

    wandb_logger = Logger(
        f"Residual_SoftHebb-Surround/HardWTA/Cos-Instar", project='
            Final-HebbianCNN', model=model)
    logger = wandb_logger.get_logger()
    num_parameters = sum(p.numel() for p in model.parameters() if p.
        requires_grad)
    print(f"Parameter Count Total: {num_parameters}")

    # unsup_optimizer = TensorLRSGD([
    #     {"params": model.conv1.parameters(), "lr": 0.08},
    #     {"params": model.res1.parameters(), "lr": 0.005},
    #     {"params": model.res2.parameters(), "lr": 0.01},
    # ], lr=0)
    # unsup_lr_scheduler = WeightNormDependentLR(unsup_optimizer,
        power_lr=0.5)
    #
    hebb_params = [
        {'params': model.conv1.parameters(), 'lr': 0.1},
        {"params": model.res1.parameters(), "lr": 0.1},
        {"params": model.res2.parameters(), "lr": 0.1},
    ]
    unsup_optimizer = optim.SGD(hebb_params, lr=0)  # The lr here
        will be overridden by the individual lrs

    sup_optimizer = optim.Adam(model.fc1.parameters(), lr=0.001)
    criterion = nn.CrossEntropyLoss()

    trn_set, tst_set, zca = data.get_data(dataset='cifar10', root='
        datasets', batch_size=64,
                                          whiten_lvl=1e-3)

    print(f'Processing Training batches: {len(trn_set)}')
```

```
    # Unsupervised training with SoftHebb
    running_loss = 0.0
    for epoch in range(1):
        print(f"Training Hebbian epoch {epoch}")
        for i, data in enumerate(trn_set, 0):
            inputs, _ = data
            inputs = inputs.to(device)
            # zero the parameter gradients
            # unsup_optimizer.zero_grad()
            # forward + update computation
            with torch.no_grad():
                outputs = model(inputs)
            # Visualize changes before updating
            if i % 200 == 0:  # Every 100 datapoint
                print(f'Saving details after batch {i}')
                plot_ltp_ltd(model.conv1, 'conv1', num_filters=10,
                   detailed_mode=True)
                plot_ltp_ltd(model.res1.conv1, 'res1.conv1',
                   num_filters=10, detailed_mode=True)
                plot_ltp_ltd(model.res1.conv2, 'res1.conv2',
                   num_filters=10, detailed_mode=True)
                plot_ltp_ltd(model.res1.conv3, 'res1.conv3',
                   num_filters=10, detailed_mode=True)
                plot_ltp_ltd(model.res2.conv1, 'res2.conv1',
                   num_filters=10, detailed_mode=True)
                plot_ltp_ltd(model.res2.conv2, 'res2.conv2',
                   num_filters=10, detailed_mode=True)
                plot_ltp_ltd(model.res2.conv3, 'res2.conv3',
                   num_filters=10, detailed_mode=True)
                model.visualize_filters('conv1')
                model.visualize_filters('res1.conv2')
                model.visualize_filters('res2.conv2')

            layers = hebbian_layers = [model.conv1,model.res1,model.
               res2]
            for layer in layers:
                if hasattr(layer, 'local_update'):
                    layer.local_update()
                else:
                    for sublayer in layer.modules():
                        if hasattr(sublayer, 'local_update'):
                            sublayer.local_update()
            # optimize
            unsup_optimizer.step()
            # unsup_lr_scheduler.step()

    print("Visualizing Filters")
    # model.visualize_filters('conv1', f'results/{"demo"}/
       demo_conv1_filters_epoch_{1}.png')
    # model.visualize_filters('res1.conv1', f'results/{"demo"}/
       demo_res1_conv1_filters_epoch_{1}.png')
    model.visualize_filters('res1.conv2', f'results/{"demo"}/
       demo_res1_conv2_filters_epoch_{1}.png')
    # model.visualize_filters('res1.conv3', f'results/{"demo"}/
       demo_res1_conv3_filters_epoch_{1}.png')
    # model.visualize_filters('res2.conv1', f'results/{"demo"}/
       demo_res2_conv1_filters_epoch_{1}.png')
```

```
191     model.visualize_filters('res2.conv2', f'results/{"demo"}/
            demo_res2_conv2_filters_epoch_{1}.png')
192     # model.visualize_filters('res2.conv3', f'results/{"demo"}/
            demo_res2_conv3_filters_epoch_{1}.png')
193     # Supervised training of classifier
194     # set requires grad false and eval mode for all modules but
            classifier
195     unsup_optimizer.zero_grad()
196     model.conv1.requires_grad = False
197     model.res1.requires_grad = False
198     model.res2.requires_grad = False
199     model.conv1.eval()
200     model.bn1.eval()
201     model.res1.eval()
202     model.res2.eval()
203
204     print("Visualizing Class separation")
205     visualize_data_clusters(tst_set, model=model, method='umap', dim
            =2)
206     print("Training Classifier")
207     for epoch in range(5):
208         model.fc1.train()
209         model.dropout.train()
210         running_loss = 0.0
211         correct = 0
212         total = 0
213         train_preds = []
214         train_labels = []
215         for i, data in enumerate(trn_set, 0):
216             inputs, labels = data
217             inputs = inputs.to(device)
218             labels = labels.to(device)
219             # zero the parameter gradients
220             sup_optimizer.zero_grad()
221             # forward + backward + optimize
222             outputs = model(inputs)
223             loss = criterion(outputs, labels)
224             loss.backward()
225             sup_optimizer.step()
226             # compute training statistics
227             running_loss += loss.item()
228             total += labels.size(0)
229             _, predicted = torch.max(outputs.data, 1)
230             correct += (predicted == labels).sum().item()
231             # For wandb logs
232             preds = torch.argmax(outputs, dim=1)
233             train_preds.append(preds)
234             train_labels.append(labels)
235
236         print(f'Accuracy of the network on the train images: {100 *
                correct // total} %')
237         print(f'[{epoch + 1}] loss: {running_loss / total:.3f}')
238
239         train_preds = torch.cat(train_preds, dim=0)
240         train_labels = torch.cat(train_labels, dim=0)
241         acc, prec, rec, f1_score, conf_matrix = calculate_metrics(
                train_preds, train_labels, 10)
```

```
        logger.log({'train_accuracy': acc, 'train_precision': prec,
            'train_recall': rec, 'train_f1_score': f1_score})
        f, ax = plt.subplots(figsize=(15, 10))
        sns.heatmap(conf_matrix.clone().detach().cpu().numpy(),
            annot=True, ax=ax)
        logger.log({"train_confusion_matrix": wandb.Image(f)})
        plt.close(f)

        # Evaluation on test set
        model.eval()
        running_loss = 0.
        correct = 0
        total = 0
        # since we're not training, we don't need to calculate the
            gradients for our outputs
        test_preds = []
        test_labels = []
        with torch.no_grad():
            for data in tst_set:
                images, labels = data
                images = images.to(device)
                labels = labels.to(device)
                # calculate outputs by running images through the
                    network
                outputs = model(images)
                # the class with the highest energy is what we
                    choose as prediction
                _, predicted = torch.max(outputs.data, 1)
                total += labels.size(0)
                correct += (predicted == labels).sum().item()
                loss = criterion(outputs, labels)
                running_loss += loss.item()
                # For wandb logs
                preds = torch.argmax(outputs, dim=1)
                test_preds.append(preds)
                test_labels.append(labels)

        print(f'Accuracy of the network on the test images: {100 *
            correct / total} %')
        print(f'test loss: {running_loss / total:.3f}')

        test_preds = torch.cat(test_preds, dim=0)
        test_labels = torch.cat(test_labels, dim=0)
        acc, prec, rec, f1_score, conf_matrix = calculate_metrics(
            test_preds, test_labels, 10)
        logger.log({'test_accuracy': acc, 'test_precision': prec, '
            test_recall': rec, 'test_f1_score': f1_score})
        f, ax = plt.subplots(figsize=(15, 10))
        sns.heatmap(conf_matrix.clone().detach().cpu().numpy(),
            annot=True, ax=ax)
        logger.log({"test_confusion_matrix": wandb.Image(f)})
        plt.close(f)

    visualize_filters(model, model.conv1)
    visualize_filters(model, model.res1.conv2)
    visualize_filters(model, model.res2.conv2)
```

A.2.13 Experiment_bp.py

```
from sklearn.manifold import TSNE
from sklearn.preprocessing import StandardScaler
import torch
import torch.nn as nn
import torch.optim as optim
import torch.optim.lr_scheduler as sched

import data
from Model_BackProp import Net_Backpropagation,
    Net_Backpropagation_depth
import numpy as np
import matplotlib.pyplot as plt
import umap
import warnings

from logger import Logger
from torchmetrics import Accuracy, Precision, Recall, F1Score,
    ConfusionMatrix
import seaborn as sns
import wandb
import pandas as pd

from receptive_fields import visualize_filters

torch.manual_seed(0)

def print_weight_statistics(layer, layer_name):
    """
    Prints statistics of the weights from a given layer.

    Args:
    layer (nn.Module): The layer to analyze
    layer_name (str): Name of the layer for printing purposes
    """
    weights = layer.weight.data

    print(f"Statistics for {layer_name}:")
    print(f"Mean: {weights.mean().item():.4f}")
    print(f"Max: {weights.max().item():.4f}")
    print(f"Min: {weights.min().item():.4f}")
    print(f"Standard Deviation: {weights.std().item():.4f}")
    print(f"Median: {weights.median().item():.4f}")
    print(f"25th Percentile: {weights.quantile(0.25).item():.4f}")
    print(f"75th Percentile: {weights.quantile(0.75).item():.4f}")
    print(f"Number of positive weights: {(weights > 0).sum().item()}
        ")
    print(f"Number of negative weights: {(weights < 0).sum().item()}
        ")
    print(f"Total number of weights: {weights.numel()}")
    print()


def plot_ltp_ltd(layer, layer_name, num_filters=10, detailed_mode=
    False):
    weights = layer.weight.data
    delta_w = layer.weight.grad

```

```
    if not detailed_mode:
        fig, ax = plt.subplots(figsize=(12, 6))

        for i in range(min(num_filters, weights.shape[0])):
            weight_change = delta_w[i].sum().item()
            color = 'green' if weight_change > 0 else 'red'
            ax.bar(i, weight_change, color=color)

        ax.axhline(y=0, color='black', linestyle='-', linewidth=0.5)
        ax.set_xlabel('Filter Index')
        ax.set_ylabel('Total Weight Change')
        ax.set_title(f'Overall LTP/LTD for first {num_filters}
            filters in {layer_name}')

        # Log the plot to wandb
        wandb.log({f"{layer_name}_Overall_LTP_LTD": wandb.Image(fig)
            })
        plt.close(fig)

    else:
        # Plot 1: Overall weight change
        fig, ax1 = plt.subplots(figsize=(12, 6))
        for i in range(min(num_filters, weights.shape[0])):
            weight_change = delta_w[i].sum().item()
            color = 'green' if weight_change > 0 else 'red'
            ax1.bar(i, weight_change, color=color)

        ax1.axhline(y=0, color='black', linestyle='-', linewidth
            =0.5)
        ax1.set_xlabel('Filter Index')
        ax1.set_ylabel('Total Weight Change')
        ax1.set_title(f'Overall LTP/LTD for first {num_filters}
            filters')

        # Log the plot to wandb
        wandb.log({f"{layer_name}_Overall_LTP_LTD": wandb.Image(fig)
            })
        plt.close(fig)

        # Plot 2: Detailed weight changes within each filter
        fig, ax2 = plt.subplots(figsize=(12, 6))
        data = []
        filter_indices = []
        for i in range(min(num_filters, weights.shape[0])):
            changes = delta_w[i].view(-1).tolist()
            data.extend(changes)
            filter_indices.extend([i] * len(changes))

        sns.violinplot(x=filter_indices, y=data, ax=ax2)
        ax2.axhline(y=0, color='black', linestyle='-', linewidth
            =0.5)
        ax2.set_xlabel('Filter Index')
        ax2.set_ylabel('Weight Change')
        ax2.set_title(f'Detailed Weight Changes within first {
            num_filters} filters')

        # Log the plot to wandb
```

```
        wandb.log({f"{layer_name}_Detailed_Weight_Changes": wandb.
            Image(fig)})
        plt.close(fig)

        # Plot 3: Detailed statistics for each filter and overall
            layer statistics
        fig, (ax3, ax4) = plt.subplots(2, 1, figsize=(12, 12),
            gridspec_kw={'height_ratios': [3, 1]})
        stats = []
        for i in range(min(num_filters, weights.shape[0])):
            filter_weights = weights[i].view(-1) #could be delta_w
            stats.append({
                'Mean': filter_weights.mean().item(),
                'Median': filter_weights.median().item(),
                'Std Dev': filter_weights.std().item(),
                '% Positive': (filter_weights > 0).float().mean().
                    item() * 100,
                '% Negative': (filter_weights < 0).float().mean().
                    item() * 100
            })
        # Per-filter statistics
        stat_df = pd.DataFrame(stats)
        sns.heatmap(stat_df.T, annot=True, cmap='coolwarm', center
            =0, ax=ax3)
        ax3.set_xlabel('Filter Index')
        ax3.set_title('Detailed Statistics for Each Filter')
        # Overall layer statistics
        all_weights = weights.view(-1)  # This includes ALL weights
            in the layer
        overall_stats = pd.DataFrame({
            'Layer Overall': {
                'Mean': all_weights.mean().item(),
                'Median': all_weights.median().item(),
                'Std Dev': all_weights.std().item(),
                '% Positive': (all_weights > 0).float().mean().item
                    () * 100,
                '% Negative': (all_weights < 0).float().mean().item
                    () * 100
            }
        })
        sns.heatmap(overall_stats, annot=True, cmap='coolwarm',
            center=0, ax=ax4)
        ax4.set_title('Overall Layer Statistics')
        plt.tight_layout()
        # Log the plot to wandb
        wandb.log({f"{layer_name}_Weight_Statistics": wandb.Image(
            fig)})
        plt.close(fig)

        # Plot 4: LTP/LTD per Weight (mean across channels)
        num_filters_to_show = min(25, weights.shape[0])
        if weights.shape[2] == 1 and weights.shape[3] == 1:  # Check
            if kernels are 1x1
            fig, ax = plt.subplots(figsize=(20, 5))
            filter_changes = [delta_w[i].mean().item() for i in
                range(num_filters_to_show)]
            norm = plt.Normalize(vmin=min(filter_changes), vmax=max(
                filter_changes))
```

```
            colors = plt.cm.RdYlGn(norm(filter_changes))
            ax.bar(range(num_filters_to_show), filter_changes, color=colors)
            ax.set_xlabel('Filter Index')
            ax.set_ylabel('Weight Change')
            ax.set_title('LTP/LTD for 1x1 Kernels')
            ax.axhline(y=0, color='black', linestyle='--', linewidth=0.5)
            sm = plt.cm.ScalarMappable(cmap="RdYlGn", norm=norm)
            sm.set_array([])
            cbar = plt.colorbar(sm, ax=ax, orientation='horizontal', aspect=30)
            cbar.set_label('Weight Change')

        else:
            rows = int(np.ceil(np.sqrt(num_filters_to_show)))
            cols = int(np.ceil(num_filters_to_show / rows))
            fig, axes = plt.subplots(rows, cols, figsize=(20, 20))
            axes = axes.flatten()

            for i in range(num_filters_to_show):
                filter_changes = delta_w[i].mean(dim=0).cpu().numpy()  # Mean across channels
                im = axes[i].imshow(filter_changes, cmap='RdYlGn', interpolation='nearest')
                axes[i].set_title(f'Filter {i}')
                axes[i].axis('off')
                plt.colorbar(im, ax=axes[i], fraction=0.046, pad=0.04)

            for j in range(i + 1, rows * cols):
                axes[j].axis('off')

            fig.suptitle('LTP/LTD per Weight (Mean across channels)', fontsize=16)

        # Log the plot to wandb
        wandb.log({f"{layer_name}_LTP_LTD_per_Weight": wandb.Image(fig)})
        plt.close(fig)


def calculate_metrics(preds, labels, num_classes):
    if num_classes == 2:
        accuracy = Accuracy(task='binary', num_classes=num_classes).to(device)
        precision = Precision(task='binary', average='weighted', num_classes=num_classes).to(device)
        recall = Recall(task='binary', average='weighted', num_classes=num_classes).to(device)
        f1 = F1Score(task='binary', average='weighted', num_classes=num_classes).to(device)
        confusion_matrix = ConfusionMatrix(task='binary', num_classes=num_classes).to(device)
    else:
        accuracy = Accuracy(task='multiclass', num_classes=num_classes).to(device)
```

```
190         precision = Precision(task='multiclass', average='macro',
                num_classes=num_classes).to(device)
191         recall = Recall(task='multiclass', average='macro',
                num_classes=num_classes).to(device)
192         f1 = F1Score(task='multiclass', average='macro', num_classes
                =num_classes).to(device)
193         confusion_matrix = ConfusionMatrix(task='multiclass',
                num_classes=num_classes).to(device)
194
195     acc = accuracy(preds, labels)
196     prec = precision(preds, labels)
197     rec = recall(preds, labels)
198     f1_score = f1(preds, labels)
199     conf_matrix = confusion_matrix(preds, labels)
200
201     return acc, prec, rec, f1_score, conf_matrix
202
203 def visualize_data_clusters(dataloader, model=None, method='tsne',
        dim=2, perplexity=30, n_neighbors=15, min_dist=0.1,
204                             n_components=2, random_state=42):
205     device = torch.device("cuda" if torch.cuda.is_available() else "
            cpu")
206     features_list = []
207     labels_list = []
208     if model is not None:
209         model.eval()
210     with torch.no_grad():
211         for data, labels in dataloader:
212             data = data.to(device)
213             if model is not None:
214                 if hasattr(model, 'features_extract'):
215                     features = model.features_extract(data)
216                 else:
217                     features = model.conv1(data)
218             else:
219                 features = data
220             features = features.view(features.size(0), -1).cpu().
                    numpy()
221             features_list.append(features)
222             labels_list.append(labels.numpy())
223     features = np.vstack(features_list)
224     labels = np.concatenate(labels_list)
225     # Normalize features
226     scaler = StandardScaler()
227     features_normalized = scaler.fit_transform(features)
228     # Apply dimensionality reduction
229     if method == 'tsne':
230         reducer = TSNE(n_components=dim, perplexity=perplexity,
                n_iter=1000, random_state=random_state)
231     elif method == 'umap':
232         reducer = umap.UMAP(n_neighbors=n_neighbors, min_dist=
                min_dist, n_components=dim, random_state=random_state)
233     else:
234         raise ValueError("Method must be either 'tsne' or 'umap'")
235     projected_data = reducer.fit_transform(features_normalized)
236     # Plotting
237     if dim == 2:
238         fig = plt.figure(figsize=(12, 10))
```

```
        scatter = plt.scatter(projected_data[:, 0], projected_data
            [:, 1], c=labels, alpha=0.5, cmap='tab10')
        plt.colorbar(scatter, label='Class Labels')
        plt.title(f'CIFAR-10 Data Clusters using {method.upper()} (2
            D)')
        plt.xlabel(f'{method.upper()} Component 1')
        plt.ylabel(f'{method.upper()} Component 2')
    elif dim == 3:
        fig = plt.figure(figsize=(12, 10))
        ax = fig.add_subplot(111, projection='3d')
        scatter = ax.scatter(projected_data[:, 0], projected_data[:,
            1], projected_data[:, 2], c=labels, alpha=0.5,
                             cmap='tab10')
        fig.colorbar(scatter, label='Class Labels')
        ax.set_title(f'CIFAR-10 Data Clusters using {method.upper()}
            (3D)')
        ax.set_xlabel(f'{method.upper()} Component 1')
        ax.set_ylabel(f'{method.upper()} Component 2')
        ax.set_zlabel(f'{method.upper()} Component 3')
    else:
        raise ValueError("dim must be either 2 or 3")

    wandb.log({"Class separation": wandb.Image(fig)})
    plt.close(fig)

# Main file to train backpropgation network
# TODO: Same visualisation code as Hebbian is not fully implemented

if __name__ == "__main__":

    device = torch.device('cuda:0')
    model = Net_Backpropagation()
    model.to(device)

    wandb_logger = Logger(
        f"SoftHebb-BackPropagation-Init",
        project='Final-HebbianCNN', model=model)
    logger = wandb_logger.get_logger()
    num_parameters = sum(p.numel() for p in model.parameters() if p.
        requires_grad)
    print(f"Parameter Count Total: {num_parameters}")

    sup_optimizer = optim.Adam(model.parameters(), lr=0.01)
    criterion = nn.CrossEntropyLoss()

    trn_set, tst_set, zca = data.get_data(dataset='cifar10', root='
        datasets', batch_size=64,
                                          whiten_lvl=None)

    print(f'Processing Training batches: {len(trn_set)}')
    for epoch in range(5):
        print(f"Training BP epoch {epoch}")
        model.train()
        running_loss = 0.0
        correct = 0
        total = 0
        train_preds = []
        train_labels = []
```

```
        for i, data in enumerate(trn_set, 0):
            inputs, labels = data
            inputs = inputs.to(device)
            labels = labels.to(device)
            # zero the parameter gradients
            sup_optimizer.zero_grad()
            # forward + backward + optimize
            outputs = model(inputs)
            loss = criterion(outputs, labels)
            loss.backward()

            if i % 200 == 0:  # Every 100 datapoint
                print(f'Saving details after batch {i}')
                plot_ltp_ltd(model.conv1, 'conv1', num_filters=10,
                    detailed_mode=True)
                plot_ltp_ltd(model.conv2, 'conv2', num_filters=10,
                    detailed_mode=True)
                # plot_ltp_ltd(model.conv_point2, 'conv_point2',
                    num_filters=10, detailed_mode=True)
                model.visualize_filters('conv1')
                model.visualize_filters('conv2')
                # model.visualize_filters('conv_point2')

            sup_optimizer.step()
            # compute training statistics
            running_loss += loss.item()
            total += labels.size(0)
            _, predicted = torch.max(outputs.data, 1)
            correct += (predicted == labels).sum().item()
            # For wandb logs
            preds = torch.argmax(outputs, dim=1)
            train_preds.append(preds)
            train_labels.append(labels)

        # print("Visualizing Receptive fields")
        # visualize_filters(model, model.conv1, num_filters=25)
        # visualize_filters(model, model.conv2, num_filters=25)
        # visualize_filters(model, model.conv3, num_filters=25)

        print(f'Accuracy of the network on the train images: {100 *
            correct // total} %')
        print(f'[{epoch + 1}] loss: {running_loss / total:.3f}')

        train_preds = torch.cat(train_preds, dim=0)
        train_labels = torch.cat(train_labels, dim=0)
        acc, prec, rec, f1_score, conf_matrix = calculate_metrics(
            train_preds, train_labels, 10)
        logger.log({'train_accuracy': acc, 'train_precision': prec,
            'train_recall': rec, 'train_f1_score': f1_score})
        f, ax = plt.subplots(figsize=(15, 10))
        sns.heatmap(conf_matrix.clone().detach().cpu().numpy(),
            annot=True, ax=ax)
        logger.log({"train_confusion_matrix": wandb.Image(f)})
        plt.close(f)

        # Evaluation on test set
        model.eval()
        running_loss = 0.
```

```
342         correct = 0
343         total = 0
344         # since we're not training, we don't need to calculate the
                gradients for our outputs
345         test_preds = []
346         test_labels = []
347         with torch.no_grad():
348             for data in tst_set:
349                 images, labels = data
350                 images = images.to(device)
351                 labels = labels.to(device)
352                 # calculate outputs by running images through the
                        network
353                 outputs = model(images)
354                 # the class with the highest energy is what we
                        choose as prediction
355                 _, predicted = torch.max(outputs.data, 1)
356                 total += labels.size(0)
357                 correct += (predicted == labels).sum().item()
358                 loss = criterion(outputs, labels)
359                 running_loss += loss.item()
360                 # For wandb logs
361                 preds = torch.argmax(outputs, dim=1)
362                 test_preds.append(preds)
363                 test_labels.append(labels)
364
365         print(f'Accuracy of the network on the test images: {100 *
                correct / total} %')
366         print(f'test loss: {running_loss / total:.3f}')
367
368         test_preds = torch.cat(test_preds, dim=0)
369         test_labels = torch.cat(test_labels, dim=0)
370         acc, prec, rec, f1_score, conf_matrix = calculate_metrics(
                test_preds, test_labels, 10)
371         logger.log({'test_accuracy': acc, 'test_precision': prec, '
                test_recall': rec, 'test_f1_score': f1_score})
372         f, ax = plt.subplots(figsize=(15, 10))
373         sns.heatmap(conf_matrix.clone().detach().cpu().numpy(),
                annot=True, ax=ax)
374         logger.log({"test_confusion_matrix": wandb.Image(f)})
375         plt.close(f)
376
377     print("Visualizing Filters")
378     model.visualize_filters('conv1', f'results/{"demo"}/
            demo_conv1_filters_epoch_{1}.png')
379     model.visualize_filters('conv2', f'results/{"demo"}/
            demo_conv2_filters_epoch_{1}.png')
380     model.visualize_filters('conv3', f'results/{"demo"}/
            demo_conv3_filters_epoch_{1}.png')
381     # model.visualize_filters('conv_point2', f'results/{"demo"}/
            demo_conv_point2_filters_epoch_{1}.png')
382     print("Weight statistics")
383     print_weight_statistics(model.conv1, 'conv1')
384     print_weight_statistics(model.conv2, 'conv2')
385     print_weight_statistics(model.conv3, 'conv3')
386     # print_weight_statistics(model.conv_point2, 'conv3')
387
388     print("Visualizing Class separation")
```

```
    visualize_data_clusters(tst_set, model=model, method='umap', dim=2)

    print("Visualizing Receptive fields")
    visualize_filters(model, model.conv1, num_filters=25)
    visualize_filters(model, model.conv2, num_filters=25)
    visualize_filters(model, model.conv3, num_filters=25)
```

A.2.14 Visualizer.py

```
import torch
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import wandb
import pandas as pd
import umap
from sklearn.manifold import TSNE
from sklearn.preprocessing import StandardScaler

torch.manual_seed(0)

def print_weight_statistics(layer, layer_name):
    """
    Prints statistics of the weights from a given layer.
    Args:
    """
    weights = layer.weight.data

    print(f"Statistics for {layer_name}:")
    print(f"Mean: {weights.mean().item():.4f}")
    print(f"Max: {weights.max().item():.4f}")
    print(f"Min: {weights.min().item():.4f}")
    print(f"Standard Deviation: {weights.std().item():.4f}")
    print(f"Median: {weights.median().item():.4f}")
    print(f"25th Percentile: {weights.quantile(0.25).item():.4f}")
    print(f"75th Percentile: {weights.quantile(0.75).item():.4f}")
    print(f"Number of positive weights: {(weights >= 0).sum().item()}")
    print(f"Number of negative weights: {(weights < 0).sum().item()}")
    print(f"Total number of weights: {weights.numel()}")
    print()


# Visualise different information regarding weights and changes in weights
def plot_ltp_ltd_ex_in(layer, layer_name, num_filters=10, detailed_mode=False):
    weights = {
        'wee': layer.weight_ee.data,
        'wei': layer.weight_ei.data,
        'wie': layer.weight_ie.data
    }
    delta_w = {
        'wee': layer.delta_w_ee.data,
        'wei': layer.delta_w_ei.data,
        'wie': layer.delta_w_ie.data
```

```
    }

    if not detailed_mode:
        fig, axes = plt.subplots(1, 3, figsize=(18, 6))
        for idx, (weight_type, weight) in enumerate(weights.items()):
            ax = axes[idx]
            delta = delta_w[weight_type]
            for i in range(min(num_filters, weight.shape[0])):
                weight_change = delta[i].sum().item()
                color = 'green' if weight_change > 0 else 'red'
                ax.bar(i, weight_change, color=color)
            ax.axhline(y=0, color='black', linestyle='-', linewidth=0.5)
            ax.set_xlabel('Filter Index')
            ax.set_ylabel('Total Weight Change')
            ax.set_title(f'Overall LTP/LTD for {weight_type} in {layer_name}')
        plt.tight_layout()
        wandb.log({f"{layer_name}_Overall_LTP_LTD": wandb.Image(fig)})
        plt.close(fig)

    else:
        for weight_type in weights.keys():
            weight = weights[weight_type]
            delta = delta_w[weight_type]

            # Plot 1: Overall weight change
            fig, ax1 = plt.subplots(figsize=(12, 6))
            for i in range(min(num_filters, weight.shape[0])):
                weight_change = delta[i].sum().item()
                color = 'green' if weight_change > 0 else 'red'
                ax1.bar(i, weight_change, color=color)
            ax1.axhline(y=0, color='black', linestyle='-', linewidth=0.5)
            ax1.set_xlabel('Filter Index')
            ax1.set_ylabel('Total Weight Change')
            ax1.set_title(f'Overall LTP/LTD for {weight_type} in {layer_name}')
            wandb.log({f"{layer_name}_{weight_type}_Overall_LTP_LTD": wandb.Image(fig)})
            plt.close(fig)

            # Plot 2: Detailed weight changes within each filter
            fig, ax2 = plt.subplots(figsize=(12, 6))
            data = []
            filter_indices = []
            for i in range(min(num_filters, weight.shape[0])):
                changes = delta[i].view(-1).tolist()
                data.extend(changes)
                filter_indices.extend([i] * len(changes))
            sns.violinplot(x=filter_indices, y=data, ax=ax2)
            ax2.axhline(y=0, color='black', linestyle='-', linewidth=0.5)
            ax2.set_xlabel('Filter Index')
            ax2.set_ylabel('Weight Change')
```

```
94                ax2.set_title(f'Detailed Weight Changes for {weight_type
                      } in {layer_name}')
95                wandb.log({f"{layer_name}_{weight_type}
                      _Detailed_Weight_Changes": wandb.Image(fig)})
96                plt.close(fig)
97
98                # Plot 3: Detailed statistics for each filter and
                      overall layer statistics
99                fig, (ax3, ax4) = plt.subplots(2, 1, figsize=(12, 12),
                      gridspec_kw={'height_ratios': [3, 1]})
100               stats = []
101               for i in range(min(num_filters, weight.shape[0])):
102                    filter_weights = weight[i].view(-1)
103                    stats.append({
104                         'Mean': filter_weights.mean().item(),
105                         'Median': filter_weights.median().item(),
106                         'Std Dev': filter_weights.std().item(),
107                         '% Positive': (filter_weights >= 0).float().mean
                                ().item() * 100,
108                         '% Negative': (filter_weights < 0).float().mean
                                ().item() * 100
109                    })
110               stat_df = pd.DataFrame(stats)
111               sns.heatmap(stat_df.T, annot=True, cmap='coolwarm',
                      center=0, ax=ax3)
112               ax3.set_xlabel('Filter Index')
113               ax3.set_title(f'Detailed Statistics for Each Filter in {
                      weight_type}')
114               all_weights = weight.view(-1)
115               overall_stats = pd.DataFrame({
116                    'Layer Overall': {
117                         'Mean': all_weights.mean().item(),
118                         'Median': all_weights.median().item(),
119                         'Std Dev': all_weights.std().item(),
120                         '% Positive': (all_weights > 0).float().mean().
                                item() * 100,
121                         '% Negative': (all_weights < 0).float().mean().
                                item() * 100
122                    }
123               })
124               sns.heatmap(overall_stats, annot=True, cmap='coolwarm',
                      center=0, ax=ax4)
125               ax4.set_title(f'Overall Layer Statistics for {
                      weight_type}')
126               plt.tight_layout()
127               wandb.log({f"{layer_name}_{weight_type}
                      _Weight_Statistics": wandb.Image(fig)})
128               plt.close(fig)
129
130               # Plot 4: LTP/LTD per Weight (mean across channels)
131               num_filters_to_show = min(25, weight.shape[0])
132               if weight.shape[2] == 1 and weight.shape[3] == 1:  #
                      Check if kernels are 1x1
133                    fig, ax = plt.subplots(figsize=(20, 5))
134                    filter_changes = [delta[i].mean().item() for i in
                           range(num_filters_to_show)]
135                    norm = plt.Normalize(vmin=min(filter_changes), vmax=
                           max(filter_changes))
```

```
                colors = plt.cm.RdYlGn(norm(filter_changes))
                ax.bar(range(num_filters_to_show), filter_changes,
                    color=colors)
                ax.set_xlabel('Filter Index')
                ax.set_ylabel('Weight Change')
                ax.set_title(f'LTP/LTD for 1x1 Kernels in {
                    weight_type}')
                ax.axhline(y=0, color='black', linestyle='--',
                    linewidth=0.5)
                sm = plt.cm.ScalarMappable(cmap="RdYlGn", norm=norm)
                sm.set_array([])
                cbar = plt.colorbar(sm, ax=ax, orientation='
                    horizontal', aspect=30)
                cbar.set_label('Weight Change')
            else:
                rows = int(np.ceil(np.sqrt(num_filters_to_show)))
                cols = int(np.ceil(num_filters_to_show / rows))
                fig, axes = plt.subplots(rows, cols, figsize=(20,
                    20))
                axes = axes.flatten()
                for i in range(num_filters_to_show):
                    filter_changes = delta[i].mean(dim=0).cpu().
                        numpy()  # Mean across channels
                    im = axes[i].imshow(filter_changes, cmap='RdYlGn
                        ', interpolation='nearest')
                    axes[i].set_title(f'Filter {i}')
                    axes[i].axis('off')
                    plt.colorbar(im, ax=axes[i], fraction=0.046, pad
                        =0.04)
                for j in range(i + 1, rows * cols):
                    axes[j].axis('off')
                fig.suptitle(f'LTP/LTD per Weight (Mean across
                    channels) for {weight_type}', fontsize=16)
            wandb.log({f"{layer_name}_{weight_type}
                _LTP_LTD_per_Weight": wandb.Image(fig)})
            plt.close(fig)

            # Plot 5: Detailed statistics for each filter and
                overall layer statistics: delta_w
            fig, (ax3, ax4) = plt.subplots(2, 1, figsize=(12, 12),
                gridspec_kw={'height_ratios': [3, 1]})
            stats = []
            for i in range(min(num_filters, weight.shape[0])):
                filter_weights = delta[i].view(-1)
                stats.append({
                    'Mean': filter_weights.mean().item(),
                    'Median': filter_weights.median().item(),
                    'Std Dev': filter_weights.std().item(),
                    '% Positive': (filter_weights > 0).float().mean
                        ().item() * 100,
                    '% Negative': (filter_weights < 0).float().mean
                        ().item() * 100
                })
            stat_df = pd.DataFrame(stats)
            sns.heatmap(stat_df.T, annot=True, cmap='coolwarm',
                center=0, ax=ax3)
            ax3.set_xlabel('Filter Index')
```

```
178             ax3.set_title(f'Detailed Statistics for Each Filter in {
                    weight_type}')
179             all_weights = delta.view(-1)
180             overall_stats = pd.DataFrame({
181                 'Layer Overall': {
182                     'Mean': all_weights.mean().item(),
183                     'Median': all_weights.median().item(),
184                     'Std Dev': all_weights.std().item(),
185                     '% Positive': (all_weights > 0).float().mean().
                            item() * 100,
186                     '% Negative': (all_weights < 0).float().mean().
                            item() * 100
187                 }
188             })
189             sns.heatmap(overall_stats, annot=True, cmap='coolwarm',
                    center=0, ax=ax4)
190             ax4.set_title(f'Overall Layer Statistics for Delta_w in
                    {weight_type}')
191             plt.tight_layout()
192             wandb.log({f"{layer_name}_{weight_type}
                    _Delta_w_Statistics": wandb.Image(fig)})
193             plt.close(fig)
194 
195             # Plot 6: Weight distribution plots
196             fig, ax6 = plt.subplots(figsize=(12, 6))
197             sns.histplot(weight.view(-1).cpu().numpy(), bins=50, kde
                    =True, ax=ax6)
198             ax6.set_xlabel('Weight Value')
199             ax6.set_ylabel('Frequency')
200             ax6.set_title(f'Weight Distribution in {layer_name} for
                    {weight_type}')
201             wandb.log({f"{layer_name}_{weight_type}
                    _Weight_Distribution": wandb.Image(fig)})
202             plt.close(fig)
203 
204 def plot_ltp_ltd(layer, layer_name, num_filters=10, detailed_mode=
        False):
205     weights = layer.weight.data
206     delta_w = layer.delta_w.data
207     # print(f"Layer {layer_name}")
208     # print(delta_w.mean())
209     # print(delta_w.max())
210 
211     if not detailed_mode:
212         fig, ax = plt.subplots(figsize=(12, 6))
213 
214         for i in range(min(num_filters, weights.shape[0])):
215             weight_change = delta_w[i].sum().item()
216             color = 'green' if weight_change > 0 else 'red'
217             ax.bar(i, weight_change, color=color)
218 
219         ax.axhline(y=0, color='black', linestyle='-', linewidth=0.5)
220         ax.set_xlabel('Filter Index')
221         ax.set_ylabel('Total Weight Change')
222         ax.set_title(f'Overall LTP/LTD for first {num_filters}
                filters in {layer_name}')
223 
224         # Log the plot to wandb
```

```
225             wandb.log({f"{layer_name}_Overall_LTP_LTD": wandb.Image(fig)
                    })
226             plt.close(fig)
227 
228         else:
229             # Plot 1: Overall weight change
230             fig, ax1 = plt.subplots(figsize=(12, 6))
231             for i in range(min(num_filters, weights.shape[0])):
232                 weight_change = delta_w[i].sum().item()
233                 color = 'green' if weight_change > 0 else 'red'
234                 ax1.bar(i, weight_change, color=color)
235 
236             ax1.axhline(y=0, color='black', linestyle='-', linewidth
                    =0.5)
237             ax1.set_xlabel('Filter Index')
238             ax1.set_ylabel('Total Weight Change')
239             ax1.set_title(f'Overall LTP/LTD for first {num_filters}
                    filters')
240 
241             # Log the plot to wandb
242             wandb.log({f"{layer_name}_Overall_LTP_LTD": wandb.Image(fig)
                    })
243             plt.close(fig)
244 
245             # Plot 2: Detailed weight changes within each filter
246             fig, ax2 = plt.subplots(figsize=(12, 6))
247             data = []
248             filter_indices = []
249             for i in range(min(num_filters, weights.shape[0])):
250                 changes = delta_w[i].view(-1).tolist()
251                 data.extend(changes)
252                 filter_indices.extend([i] * len(changes))
253 
254             sns.violinplot(x=filter_indices, y=data, ax=ax2)
255             ax2.axhline(y=0, color='black', linestyle='-', linewidth
                    =0.5)
256             ax2.set_xlabel('Filter Index')
257             ax2.set_ylabel('Weight Change')
258             ax2.set_title(f'Detailed Weight Changes within first {
                    num_filters} filters')
259 
260             # Log the plot to wandb
261             wandb.log({f"{layer_name}_Detailed_Weight_Changes": wandb.
                    Image(fig)})
262             plt.close(fig)
263 
264             # Plot 3: Detailed statistics for each filter and overall
                    layer statistics
265             fig, (ax3, ax4) = plt.subplots(2, 1, figsize=(12, 12),
                    gridspec_kw={'height_ratios': [3, 1]})
266             stats = []
267             for i in range(min(num_filters, weights.shape[0])):
268                 filter_weights = weights[i].view(-1) #could be delta_w
269                 stats.append({
270                     'Mean': filter_weights.mean().item(),
271                     'Median': filter_weights.median().item(),
272                     'Std Dev': filter_weights.std().item(),
```

```
273                 '% Positive': (filter_weights > 0).float().mean().
                        item() * 100,
274                 '% Negative': (filter_weights < 0).float().mean().
                        item() * 100
275             })
276         # Per-filter statistics
277         stat_df = pd.DataFrame(stats)
278         sns.heatmap(stat_df.T, annot=True, cmap='coolwarm', center
                =0, ax=ax3)
279         ax3.set_xlabel('Filter Index')
280         ax3.set_title('Detailed Statistics for Each Filter')
281         # Overall layer statistics
282         all_weights = weights.view(-1)  # This includes ALL weights
                in the layer
283         overall_stats = pd.DataFrame({
284             'Layer Overall': {
285                 'Mean': all_weights.mean().item(),
286                 'Median': all_weights.median().item(),
287                 'Std Dev': all_weights.std().item(),
288                 '% Positive': (all_weights > 0).float().mean().item
                        () * 100,
289                 '% Negative': (all_weights < 0).float().mean().item
                        () * 100
290             }
291         })
292         sns.heatmap(overall_stats, annot=True, cmap='coolwarm',
                center=0, ax=ax4)
293         ax4.set_title('Overall Layer Statistics')
294         plt.tight_layout()
295         # Log the plot to wandb
296         wandb.log({f"{layer_name}_Weight_Statistics": wandb.Image(
                fig)})
297         plt.close(fig)
298
299         # Plot 4: LTP/LTD per Weight (mean across channels)
300         num_filters_to_show = min(25, weights.shape[0])
301         if weights.shape[2] == 1 and weights.shape[3] == 1:  # Check
                if kernels are 1x1
302             fig, ax = plt.subplots(figsize=(20, 5))
303             filter_changes = [delta_w[i].mean().item() for i in
                    range(num_filters_to_show)]
304             norm = plt.Normalize(vmin=min(filter_changes), vmax=max(
                    filter_changes))
305             colors = plt.cm.RdYlGn(norm(filter_changes))
306             ax.bar(range(num_filters_to_show), filter_changes, color
                    =colors)
307             ax.set_xlabel('Filter Index')
308             ax.set_ylabel('Weight Change')
309             ax.set_title('LTP/LTD for 1x1 Kernels')
310             ax.axhline(y=0, color='black', linestyle='--', linewidth
                    =0.5)
311             sm = plt.cm.ScalarMappable(cmap="RdYlGn", norm=norm)
312             sm.set_array([])
313             cbar = plt.colorbar(sm, ax=ax, orientation='horizontal',
                        aspect=30)
314             cbar.set_label('Weight Change')
315
316         else:
```

```
            rows = int(np.ceil(np.sqrt(num_filters_to_show)))
            cols = int(np.ceil(num_filters_to_show / rows))
            fig, axes = plt.subplots(rows, cols, figsize=(20, 20))
            axes = axes.flatten()

            for i in range(num_filters_to_show):
                filter_changes = delta_w[i].mean(dim=0).cpu().numpy()  # Mean across channels
                im = axes[i].imshow(filter_changes, cmap='RdYlGn', interpolation='nearest')
                axes[i].set_title(f'Filter {i}')
                axes[i].axis('off')
                plt.colorbar(im, ax=axes[i], fraction=0.046, pad=0.04)

            for j in range(i + 1, rows * cols):
                axes[j].axis('off')

            fig.suptitle('LTP/LTD per Weight (Mean across channels)', fontsize=16)
        # Log the plot to wandb
        wandb.log({f"{layer_name}_LTP_LTD_per_Weight": wandb.Image(fig)})
        plt.close(fig)

        # Plot 5: Detailed statistics for each filter and overall layer statistics: delta_w
        fig, (ax3, ax4) = plt.subplots(2, 1, figsize=(12, 12), gridspec_kw={'height_ratios': [3, 1]})
        stats = []
        for i in range(min(num_filters, weights.shape[0])):
            filter_weights = delta_w[i].view(-1)  # could be delta_w
            stats.append({
                'Mean': filter_weights.mean().item(),
                'Median': filter_weights.median().item(),
                'Std Dev': filter_weights.std().item(),
                '% Positive': (filter_weights > 0).float().mean().item() * 100,
                '% Negative': (filter_weights < 0).float().mean().item() * 100
            })
        # Per-filter statistics
        stat_df = pd.DataFrame(stats)
        sns.heatmap(stat_df.T, annot=True, cmap='coolwarm', center=0, ax=ax3)
        ax3.set_xlabel('Filter Index')
        ax3.set_title('Detailed Statistics for Each Filter')
        # Overall layer statistics
        all_weights = delta_w.view(-1)  # This includes ALL weights in the layer
        overall_stats = pd.DataFrame({
            'Layer Overall': {
                'Mean': all_weights.mean().item(),
                'Median': all_weights.median().item(),
                'Std Dev': all_weights.std().item(),
                '% Positive': (all_weights > 0).float().mean().item() * 100,
```

```
                '% Negative': (all_weights < 0).float().mean().item() * 100
            }
        })
        sns.heatmap(overall_stats, annot=True, cmap='coolwarm', center=0, ax=ax4)
        ax4.set_title('Overall Layer Statistics for Delta_w')
        plt.tight_layout()
        # Log the plot to wandb
        wandb.log({f"{layer_name}_Delta_w_Statistics": wandb.Image(fig)})
        plt.close(fig)

        # Plot 6: Weight distribution plots to understand connectivity of weights better
        fig, ax6 = plt.subplots(figsize=(12, 6))
        sns.histplot(weights.view(-1).cpu().numpy(), bins=50, kde=True, ax=ax6)
        ax6.set_xlabel('Weight Value')
        ax6.set_ylabel('Frequency')
        ax6.set_title(f'Weight Distribution in {layer_name}')
        # Log the plot to wandb
        wandb.log({f"{layer_name}_Weight_Distribution": wandb.Image(fig)})
        plt.close(fig)


# Visualise class separation
def visualize_data_clusters(dataloader, model=None, method='tsne', dim=2, perplexity=30, n_neighbors=15, min_dist=0.1,
                            n_components=2, random_state=42):
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    features_list = []
    labels_list = []
    if model is not None:
        model.eval()
    with torch.no_grad():
        for data, labels in dataloader:
            data = data.to(device)
            if model is not None:
                if hasattr(model, 'features_extract'):
                    features = model.features_extract(data)
                else:
                    features = model.conv1(data)
            else:
                features = data
            features = features.view(features.size(0), -1).cpu().numpy()
            features_list.append(features)
            labels_list.append(labels.numpy())
    features = np.vstack(features_list)
    labels = np.concatenate(labels_list)
    # Normalize features
    scaler = StandardScaler()
    features_normalized = scaler.fit_transform(features)
    # Apply dimensionality reduction
    if method == 'tsne':
```

```
411         reducer = TSNE(n_components=dim, perplexity=perplexity,
                n_iter=1000, random_state=random_state)
412     elif method == 'umap':
413         reducer = umap.UMAP(n_neighbors=n_neighbors, min_dist=
                min_dist, n_components=dim, random_state=random_state)
414     else:
415         raise ValueError("Method must be either 'tsne' or 'umap'")
416     projected_data = reducer.fit_transform(features_normalized)
417     # Plotting
418     if dim == 2:
419         fig = plt.figure(figsize=(12, 10))
420         scatter = plt.scatter(projected_data[:, 0], projected_data
                [:, 1], c=labels, alpha=0.5, cmap='tab10')
421         plt.colorbar(scatter, label='Class Labels')
422         plt.title(f'CIFAR-10 Data Clusters using {method.upper()} (2
                D)')
423         plt.xlabel(f'{method.upper()} Component 1')
424         plt.ylabel(f'{method.upper()} Component 2')
425     elif dim == 3:
426         fig = plt.figure(figsize=(12, 10))
427         ax = fig.add_subplot(111, projection='3d')
428         scatter = ax.scatter(projected_data[:, 0], projected_data[:,
                1], projected_data[:, 2], c=labels, alpha=0.5,
429                              cmap='tab10')
430         fig.colorbar(scatter, label='Class Labels')
431         ax.set_title(f'CIFAR-10 Data Clusters using {method.upper()}
                (3D)')
432         ax.set_xlabel(f'{method.upper()} Component 1')
433         ax.set_ylabel(f'{method.upper()} Component 2')
434         ax.set_zlabel(f'{method.upper()} Component 3')
435     else:
436         raise ValueError("dim must be either 2 or 3")
437 
438     wandb.log({"Class separation": wandb.Image(fig)})
439     plt.close(fig)
```

A.2.15 receptive_field.py

```
1  import matplotlib.pyplot as plt
2  import torch
3  from torch import nn, optim
4  import torch.nn.functional as F
5  from hebb import HebbianConv2d
6  from hebb_depthwise import HebbianDepthConv2d
7 
8  # Manually change if using Dale network
9  # from hebb_abs import HebbianConv2d
10 # from hebb_abs_depthwise import HebbianDepthConv2d
11 import wandb
12 
13 # Code to visualise receptive fields
14 
15 def get_partial_model(model, target_layer):
16     layers = []
17     for layer in model.children():
18         layers.append(layer)
19         if layer == target_layer:
20             break
```

```
    return torch.nn.Sequential(*layers)


def calculate_receptive_field(model, target_layer):
    current_rf = 1
    current_stride = 1

    for layer in model.children():
        if isinstance(layer, (nn.Conv2d, HebbianConv2d,
           HebbianDepthConv2d)):
            kernel_size = layer.kernel_size[0] if isinstance(layer.
               kernel_size, tuple) else layer.kernel_size
            stride = layer.stride[0] if isinstance(layer.stride,
               tuple) else layer.stride
            current_rf += (kernel_size - 1) * current_stride
            current_stride *= stride
        elif isinstance(layer, (nn.MaxPool2d, nn.AvgPool2d)):
            kernel_size = layer.kernel_size if isinstance(layer.
               kernel_size, int) else layer.kernel_size[0]
            stride = layer.stride if isinstance(layer.stride, int)
               else layer.stride[0]
            current_rf += (kernel_size - 1) * current_stride
            current_stride *= stride

        if layer == target_layer:
            break

    return current_rf

def get_layer_output(model, x, target_layer):
    """
    Forward pass through the model, stopping at the target custom
       Hebbian layer.
    """
    for layer in model.children():
        x = layer(x)  # Pass through each layer
        if layer == target_layer:
            return x  # Return the output of the target Hebbian
               layer
    raise ValueError(f"Target layer {target_layer} not found in the
       model.")

def remove_padding(model, target_layer):
    """Remove padding from layers before the target layer."""
    for layer in model.children():
        if isinstance(layer, nn.Conv2d):
            layer.padding = (0, 0)
        elif isinstance(layer, (HebbianConv2d, HebbianDepthConv2d)):
            # For custom SoftHebbConv2d layers, we need to modify
               the padding directly
            layer.padding = 0
        if isinstance(layer, (nn.MaxPool2d, nn.AvgPool2d)):
            layer.padding = (0, 0)
        if layer == target_layer:
            break
    return model

def gaussian_blur(x, kernel_size=5, sigma=1.0):
```

```
    channels = x.shape[1]
    kernel = torch.tensor([
        [1., 4., 6., 4., 1.],
        [4., 16., 24., 16., 4.],
        [6., 24., 36., 24., 6.],
        [4., 16., 24., 16., 4.],
        [1., 4., 6., 4., 1.]
    ], device=x.device).unsqueeze(0).unsqueeze(0) / 256.0
    kernel = kernel.repeat(channels, 1, 1, 1)
    padding = kernel_size // 2
    return F.conv2d(x, kernel, padding=padding, groups=channels)

class SingleMax:
    def __init__(self, max_val: float, eps: float):
        self.max_val = max_val
        self.eps = eps

    def __call__(self, outputs):
        if self.max_val is None:
            return outputs > 0
        else:
            return (self.max_val - outputs) < self.eps

class L2ProjGradientDescent:
    def __init__(self, steps, random_start=True, rel_stepsize=0.1):
        self.steps = steps
        self.random_start = random_start
        self.rel_stepsize = rel_stepsize

    def get_random_start(self, x0, epsilon):
        batch_size, c, h, w = x0.shape
        r = torch.randn(batch_size, c * h * w, device=x0.device)
        r = r / r.norm(dim=1, keepdim=True)
        r = r.view_as(x0)
        return x0 + 0.00001 * epsilon * r

    def normalize_gradient(self, grad):
        return grad / (grad.view(grad.shape[0], -1).norm(dim=1).view
           (-1, 1, 1, 1) + 1e-8)

    def project(self, x, x0, epsilon):
        delta = x - x0
        delta = epsilon * delta / delta.view(delta.shape[0], -1).
           norm(dim=1).view(-1, 1, 1, 1).clamp(min=1e-12)
        return x0 + delta

    def run(self, model, x0, target_layer, filter_idx, epsilon,
       criterion):
        x = x0.clone()
        if self.random_start:
            x = self.get_random_start(x0, epsilon)
        for _ in range(self.steps):
            x.requires_grad_(True)
            x_smooth = gaussian_blur(x, sigma=1.0)  # Apply Gaussian
                smoothing
            activation = get_layer_output(model, x, target_layer)
            loss = -activation[0, filter_idx].sum()
            if criterion(loss.item()):
```

```
                break
            grad = torch.autograd.grad(loss, x)[0]
            grad = self.normalize_gradient(grad)
            with torch.no_grad():
                x = x - self.rel_stepsize * epsilon * grad
                x = self.project(x, x0, epsilon)
                x.clamp_(0, 1)
        return x

def visualize_filters(model, layer, num_filters=25, input_shape=(1, 3, 32, 32), step_size=0.001, iterations=500,
                      random_start=True, nb_start=1, l2_norm=True, max_val=1e10, eps=1e-05, epsilons=None):
    if epsilons is None:
        epsilons = torch.tensor([255.0], device='cuda') / 255.0
    else:
        epsilons = torch.tensor(epsilons, device='cuda')
    model.eval()
    model = remove_padding(model, layer)
    receptive_field_size = calculate_receptive_field(model, layer)
    print(f"Receptive field size: {receptive_field_size}x{receptive_field_size}")
    input_shape = (1, 3, receptive_field_size, receptive_field_size)
    if hasattr(layer, 'out_channels'):
        out_channels = layer.out_channels
    else:
        raise ValueError("The target layer does not have an 'out_channels' attribute.")
    num_filters = min(num_filters, out_channels)
    filter_images = []
    criterion = SingleMax(max_val, eps)
    pgd = L2ProjGradientDescent(steps=iterations, random_start=random_start, rel_stepsize=step_size)

    for filter_idx in range(num_filters):
        best_image = None
        best_activation = float('-inf')
        for start in range(nb_start):
            x0 = torch.zeros(input_shape, device='cuda', requires_grad=True)
            optimized_image = pgd.run(model, x0, layer, filter_idx, epsilons, criterion)
            activation = -get_layer_output(model, optimized_image, layer)[0, filter_idx].sum().item()
            if activation > best_activation:
                print(f"New best Receptive Field found for filter {filter_idx} in Reboot {start}")
                best_activation = activation
                best_image = optimized_image
        optimized_image = best_image.cpu().squeeze(0).permute(1, 2, 0)
        optimized_image = (optimized_image - optimized_image.min()) / (optimized_image.max() - optimized_image.min())
        filter_images.append(optimized_image)
    # Plot the filter visualizations in a grid
    grid_size = int(num_filters ** 0.5) + (1 if num_filters ** 0.5 % 1 > 0 else 0)
    fig, axes = plt.subplots(grid_size, grid_size, figsize=(20, 20))
```

```
    axes = axes.flatten()
    for i in range(num_filters):
        axes[i].imshow(filter_images[i].numpy())
        axes[i].set_title(f'Filter {i + 1}')
        axes[i].axis('off')
    # Turn off unused subplots
    for j in range(num_filters, len(axes)):
        axes[j].axis('off')
    plt.tight_layout()
    wandb.log({"Rceptive Fields": wandb.Image(fig)})
    plt.close(fig)
```

A.2.16 receptive_field_residual.py

```
import matplotlib.pyplot as plt
import torch
from torch import nn
import torch.nn.functional as F
from hebb import HebbianConv2d
from hebb_depthwise import HebbianDepthConv2d
from model_residual import HebbianResidualBlock
import wandb

# Code to visualise receptive fields. Modified version of
    receptive_fields.py to work with residual blocks


def remove_padding_except_conv2(model):
    for module in model.modules():
        if isinstance(module, HebbianResidualBlock):
            module.conv1.padding = 0
            # Keep padding for conv2
            module.conv3.padding = 0
            if isinstance(module.shortcut, nn.Sequential):
                for layer in module.shortcut:
                    if isinstance(layer, (nn.Conv2d, HebbianConv2d))
                        :
                        layer.padding = 0
        elif isinstance(module, (nn.Conv2d, HebbianConv2d,
            HebbianDepthConv2d)):
            module.padding = 0
        elif isinstance(module, (nn.MaxPool2d, nn.AvgPool2d)):
            module.padding = 0

def get_partial_model(model, target_layer):
    layers = []
    for name, module in model.named_modules():
        if isinstance(module, (nn.Conv2d, HebbianConv2d,
            HebbianDepthConv2d, HebbianResidualBlock, nn.MaxPool2d,
            nn.BatchNorm2d)):
            if isinstance(module, HebbianResidualBlock):
                layers.append((f"{name}.bn1", module.bn1))
                layers.append((f"{name}.conv1", module.conv1))
                layers.append((f"{name}.bn2", module.bn2))
                layers.append((f"{name}.conv2", module.conv2))
                layers.append((f"{name}.bn3", module.bn3))
                layers.append((f"{name}.conv3", module.conv3))
                if isinstance(module.shortcut, nn.Sequential):
```

```
                    for i, shortcut_layer in enumerate(module.shortcut):
                        layers.append((f"{name}.shortcut.{i}", shortcut_layer))
            else:
                layers.append((name, module))
        if module == target_layer:
            break
    return layers


def calculate_receptive_field(model, target_layer):
    current_rf = 1
    current_stride = 1

    print(f"Calculating receptive field for {target_layer}")
    print(f"Initial RF: {current_rf}, Initial stride: {current_stride}")

    for name, module in model.named_modules():
        if isinstance(module, (nn.Conv2d, HebbianConv2d, HebbianDepthConv2d)):
            kernel_size = module.kernel_size[0] if isinstance(module.kernel_size, tuple) else module.kernel_size
            stride = module.stride[0] if isinstance(module.stride, tuple) else module.stride
            prev_rf = current_rf
            current_rf += (kernel_size - 1) * current_stride
            prev_stride = current_stride
            current_stride *= stride
            print(f"Layer: {name}, Type: Conv, Kernel: {kernel_size}, Stride: {stride}")
            print(f"  RF: {prev_rf} -> {current_rf}, Stride: {prev_stride} -> {current_stride}")
        elif isinstance(module, nn.MaxPool2d):
            kernel_size = module.kernel_size if isinstance(module.kernel_size, int) else module.kernel_size[0]
            stride = module.stride if isinstance(module.stride, int) else module.stride[0]
            prev_rf = current_rf
            current_rf += (kernel_size - 1) * current_stride
            prev_stride = current_stride
            current_stride *= stride
            print(f"Layer: {name}, Type: MaxPool, Kernel: {kernel_size}, Stride: {stride}")
            print(f"  RF: {prev_rf} -> {current_rf}, Stride: {prev_stride} -> {current_stride}")
        else:
            print(f"Layer: {name}, Type: {type(module).__name__} (no effect on RF)")

        if module == target_layer:
            print(f"Reached target layer. Final receptive field: {current_rf}x{current_rf}")
            return current_rf

    print(f"Target layer not found. Final receptive field: {current_rf}x{current_rf}")
```

```
    return current_rf


def get_layer_output(model, x, target_layer, debug=False):
    if debug:
        print(f"Initial input shape: {x.shape}")

    for name, module in model.named_children():
        if debug:
            print(f"Processing module: {name}")

        if isinstance(module, HebbianResidualBlock):
            if debug:
                print(f"Entering {name}, input shape: {x.shape}")
            residual = x
            out = module.activ(module.conv1(module.bn1(x)))
            if debug:
                print(f"  After conv1: {out.shape}")
            out = module.activ(module.conv2(module.bn2(out)))
            if debug:
                print(f"  After conv2: {out.shape}")
            out = module.conv3(module.bn3(out))
            if debug:
                print(f"  After conv3: {out.shape}")
            shortcut_output = module.shortcut(residual)
            if debug:
                print(f"  Shortcut output: {shortcut_output.shape}")

            # Ensure out and shortcut_output have the same size
            if out.size() != shortcut_output.size():
                min_size = min(out.size(2), shortcut_output.size(2))
                out = out[:, :, :min_size, :min_size]
                shortcut_output = shortcut_output[:, :, :min_size, :
                    min_size]

            x = module.activ(out + shortcut_output)
            if debug:
                print(f"Exiting {name}, output shape: {x.shape}")
        elif isinstance(module, (nn.Conv2d, HebbianConv2d,
            HebbianDepthConv2d, nn.MaxPool2d, nn.AvgPool2d)):
            if debug:
                print(f"Entering {name}, input shape: {x.shape}")
            x = module(x)
            if debug:
                print(f"Exiting {name}, output shape: {x.shape}")
        elif isinstance(module, nn.BatchNorm2d):
            if debug:
                print(
                    f"Entering {name} (BatchNorm2d), input shape: {x
                        .shape}, expected features: {module.
                        num_features}")
            x = module(x)
            if debug:
                print(f"Exiting {name} (BatchNorm2d), output shape:
                    {x.shape}")

        if module == target_layer:
            if debug:
```

```
                print(f"Reached target layer {name}, output shape: {
                    x.shape}")
            return x

        # If the target is inside a HebbianResidualBlock, we need to
            look inside it
        if isinstance(module, HebbianResidualBlock):
            for sub_name, sub_module in module.named_children():
                if sub_module == target_layer:
                    if debug:
                        print(f"Reached target layer {name}.{
                            sub_name}, output shape: {x.shape}")
                    return x

    raise ValueError(f"Target layer {target_layer} not found in the
        model.")


def gaussian_blur(x, kernel_size=5, sigma=1.0):
    channels = x.shape[1]
    kernel = torch.tensor([
        [1., 4., 6., 4., 1.],
        [4., 16., 24., 16., 4.],
        [6., 24., 36., 24., 6.],
        [4., 16., 24., 16., 4.],
        [1., 4., 6., 4., 1.]
    ], device=x.device).unsqueeze(0).unsqueeze(0) / 256.0
    kernel = kernel.repeat(channels, 1, 1, 1)
    padding = kernel_size // 2
    return F.conv2d(x, kernel, padding=padding, groups=channels)

class SingleMax:
    def __init__(self, max_val: float, eps: float):
        self.max_val = max_val
        self.eps = eps

    def __call__(self, outputs):
        if self.max_val is None:
            return outputs > 0
        else:
            return (self.max_val - outputs) < self.eps

class L2ProjGradientDescent:
    def __init__(self, steps, random_start=True, rel_stepsize=0.1):
        self.steps = steps
        self.random_start = random_start
        self.rel_stepsize = rel_stepsize

    def get_random_start(self, x0, epsilon):
        batch_size, c, h, w = x0.shape
        r = torch.randn(batch_size, c * h * w, device=x0.device)
        r = r / r.norm(dim=1, keepdim=True)
        r = r.view_as(x0)
        return x0 + 0.00001 * epsilon * r

    def normalize_gradient(self, grad):
        return grad / (grad.view(grad.shape[0], -1).norm(dim=1).view
            (-1, 1, 1, 1) + 1e-8)
```

```
189
190     def project(self, x, x0, epsilon):
191         delta = x - x0
192         delta = epsilon * delta / delta.view(delta.shape[0], -1).
                norm(dim=1).view(-1, 1, 1, 1).clamp(min=1e-12)
193         return x0 + delta
194
195     def run(self, model, x0, target_layer, filter_idx, epsilon,
            criterion, debug=False):
196         x = x0.clone()
197         if self.random_start:
198             x = self.get_random_start(x0, epsilon)
199         for _ in range(self.steps):
200             x.requires_grad_(True)
201             x_smooth = gaussian_blur(x, sigma=1.0)  # Apply Gaussian
                    smoothing
202             activation = get_layer_output(model, x, target_layer,
                    debug=debug)
203             loss = -activation[0, filter_idx].sum()
204             if criterion(loss.item()):
205                 break
206             grad = torch.autograd.grad(loss, x)[0]
207             grad = self.normalize_gradient(grad)
208             with torch.no_grad():
209                 x = x - self.rel_stepsize * epsilon * grad
210                 x = self.project(x, x0, epsilon)
211                 x.clamp_(0, 1)
212         return x
213
214
215 def visualize_filters(model, target_layer, num_filters=25,
        input_shape=(1, 3, 32, 32), step_size=0.001, iterations=500,
216                       random_start=True, nb_start=1, l2_norm=True,
                              max_val=1e10, eps=1e-05, epsilons=None,
                              debug=False):
217     if epsilons is None:
218         epsilons = torch.tensor([255.0], device='cuda') / 255.0
219     else:
220         epsilons = torch.tensor(epsilons, device='cuda')
221     model.eval()
222     remove_padding_except_conv2(model)
223     receptive_field_size = calculate_receptive_field(model,
            target_layer)
224     print(f"Receptive field size: {receptive_field_size}x{
            receptive_field_size}")
225     input_shape = (1, 3, receptive_field_size, receptive_field_size)
226
227     if isinstance(target_layer, HebbianResidualBlock):
228         out_channels = target_layer.conv3.out_channels
229     elif hasattr(target_layer, 'out_channels'):
230         out_channels = target_layer.out_channels
231     else:
232         raise ValueError("The target layer does not have an '
                out_channels' attribute.")
233
234     num_filters = min(num_filters, out_channels)
235     filter_images = []
236     criterion = SingleMax(max_val, eps)
```

```
    pgd = L2ProjGradientDescent(steps=iterations, random_start=
        random_start, rel_stepsize=step_size)

    for filter_idx in range(num_filters):
        best_image = None
        best_activation = float('-inf')
        for start in range(nb_start):
            x0 = torch.zeros(input_shape, device='cuda',
                requires_grad=True)
            optimized_image = pgd.run(model, x0, target_layer,
                filter_idx, epsilons, criterion, debug=debug)
            activation = -get_layer_output(model, optimized_image,
                target_layer, debug=debug)[0, filter_idx].sum().item
                ()
            if activation > best_activation:
                print(f"New best Receptive Field found for filter {
                    filter_idx} in Reboot {start}")
                best_activation = activation
                best_image = optimized_image
        optimized_image = best_image.cpu().squeeze(0).permute(1, 2,
            0)
        optimized_image = (optimized_image - optimized_image.min())
            / (optimized_image.max() - optimized_image.min())
        filter_images.append(optimized_image)

    # Plot the filter visualizations in a grid
    grid_size = int(num_filters ** 0.5) + (1 if num_filters ** 0.5 %
        1 > 0 else 0)
    fig, axes = plt.subplots(grid_size, grid_size, figsize=(20, 20))
    axes = axes.flatten()
    for i in range(num_filters):
        axes[i].imshow(filter_images[i].numpy())
        axes[i].set_title(f'Filter {i + 1}')
        axes[i].axis('off')
    # Turn off unused subplots
    for j in range(num_filters, len(axes)):
        axes[j].axis('off')
    plt.tight_layout()
    wandb.log({"Receptive Fields": wandb.Image(fig)})
    plt.close(fig)
```

A.2.17 params.py

```
import torch

AVAILABLE_DEVICES = ['cpu']
if torch.cuda.is_available(): AVAILABLE_DEVICES += ['cuda:{}'.format
    (d) for d in range(torch.cuda.device_count())]
DEFAULT_DEVICE = 'cuda:0' #'cpu'
NUM_WORKERS = 4
DIST_BINS = 20
```

A.3 Architectures

Table 5: SoftHebb CNN Architecture (Non-Depthwise)

Layer	Type	Output Shape	Kernel	Stride	Padding	Activation
Input	-	(3, 32, 32)	-	-	-	-
1	BatchNorm2d	(3, 32, 32)	-	-	-	-
	HebbianConv2d	(96, 32, 32)	5x5	1	2	-
	Triangle	(96, 32, 32)	-	-	-	power=0.7
	MaxPool2d	(96, 16, 16)	4x4	2	1	-
2	BatchNorm2d	(96, 16, 16)	-	-	-	-
	HebbianConv2d	(384, 16, 16)	3x3	1	1	-
	Triangle	(384, 16, 16)	-	-	-	power=1.4
	MaxPool2d	(384, 8, 8)	4x4	2	1	-
3	BatchNorm2d	(384, 8, 8)	-	-	-	-
	HebbianConv2d	(1536, 8, 8)	3x3	1	1	-
	Triangle	(1536, 8, 8)	-	-	-	power=1.0
	AvgPool2d	(1536, 4, 4)	2x2	2	0	-
Output	Linear	(10)	-	-	-	-

Table 6: HardHebb CNN Architecture (Non-Depthwise)

Layer	Type	Output Shape	Kernel	Stride	Padding	Activation
Input	-	(3, 32, 32)	-	-	-	-
1	BatchNorm2d	(3, 32, 32)	-	-	-	-
	HebbianConv2d	(96, 28, 28)	5x5	1	0	Cosine
	Triangle	(96, 28, 28)	-	-	-	power=0.7
	MaxPool2d	(96, 14, 14)	2x2	2	0	-
2	BatchNorm2d	(96, 14, 14)	-	-	-	-
	HebbianConv2d	(384, 12, 12)	3x3	1	0	Cosine
	Triangle	(384, 12, 12)	-	-	-	power=1.4
3	BatchNorm2d	(384, 12, 12)	-	-	-	-
	HebbianConv2d	(1536, 10, 10)	3x3	1	0	Cosine
	Triangle	(1536, 10, 10)	-	-	-	power=1.0
	AvgPool2d	(1536, 5, 5)	2x2	2	0	-
Output	Linear	(10)	-	-	-	-

* Cosine = Cosine Similarirty between weights and input

Table 7: Lagani Short CNN Architecture (Non-Depthwise)

Layer	Type	Output Shape	Kernel	Stride	Padding	Activation
Input	-	(3, 32, 32)	-	-	-	-
1	BatchNorm2d	(3, 32, 32)	-	-	-	-
	HebbianConv2d	(96, 28, 28)	5x5	1	0	Cosine
	Triangle	(96, 28, 28)	-	-	-	power=1.0
	MaxPool2d	(96, 14, 14)	2x2	2	0	-
2	BatchNorm2d	(96, 14, 14)	-	-	-	-
	HebbianConv2d	(128, 12, 12)	3x3	1	0	Cosine
	Triangle	(128, 12, 12)	-	-	-	power=1.0
3	BatchNorm2d	(128, 12, 12)	-	-	-	-
	HebbianConv2d	(192, 10, 10)	3x3	1	0	Cosine
	Triangle	(192, 10, 10)	-	-	-	power=1.0
	AvgPool2d	(192, 5, 5)	2x2	2	0	-
Output	Linear	(10)	-	-	-	-

Table 8: Lagani Long CNN Architecture (Non-Depthwise)

Layer	Type	Output Shape	Kernel	Stride	Padding	Activation
Input	-	(3, 32, 32)	-	-	-	-
1	BatchNorm2d	(3, 32, 32)	-	-	-	-
	HebbianConv2d	(96, 28, 28)	5x5	1	0	Cosine
	Triangle	(96, 28, 28)	-	-	-	power=1.0
	MaxPool2d	(96, 14, 14)	2x2	2	0	-
2	BatchNorm2d	(96, 14, 14)	-	-	-	-
	HebbianConv2d	(128, 12, 12)	3x3	1	0	Cosine
	Triangle	(128, 12, 12)	-	-	-	power=1.0
3	BatchNorm2d	(128, 12, 12)	-	-	-	-
	HebbianConv2d	(192, 10, 10)	3x3	1	0	Cosine
	Triangle	(192, 10, 10)	-	-	-	power=1.0
	AvgPool2d	(192, 5, 5)	2x2	2	0	-
4	BatchNorm2d	(192, 5, 5)	-	-	-	-
	HebbianConv2d	(256, 3, 3)	3x3	1	0	Cosine
	Triangle	(256, 3, 3)	-	-	-	power=1.0
Output	Linear	(10)	-	-	-	-

Table 9: SoftHebb CNN Architecture (Depthwise)

Layer	Type	Output Shape	Kernel	Stride	Padding	Activation
Input	-	(3, 32, 32)	-	-	-	-
1	BatchNorm2d	(3, 32, 32)	-	-	-	-
	HebbianConv2d	(96, 32, 32)	5x5	1	2	-
	Triangle	(96, 32, 32)	-	-	-	power=0.7
	MaxPool2d	(96, 16, 16)	4x4	2	1	-
2	BatchNorm2d	(96, 16, 16)	-	-	-	-
	HebbianDepthConv2d	(96, 16, 16)	3x3	1	1	-
	BatchNorm2d	(96, 16, 16)	-	-	-	-
	HebbianConv2d	(384, 16, 16)	1x1	1	0	-
	Triangle	(384, 16, 16)	-	-	-	power=1.4
	MaxPool2d	(384, 8, 8)	4x4	2	1	-
3	BatchNorm2d	(384, 8, 8)	-	-	-	-
	HebbianDepthConv2d	(384, 8, 8)	3x3	1	1	-
	BatchNorm2d	(384, 8, 8)	-	-	-	-
	HebbianConv2d	(1536, 8, 8)	1x1	1	0	-
	Triangle	(1536, 8, 8)	-	-	-	power=1.0
	AvgPool2d	(1536, 4, 4)	2x2	2	0	-
Output	Linear	(10)	-	-	-	-

Table 10: HardHebb CNN Architecture (Depthwise)

Layer	Type	Output Shape	Kernel	Stride	Padding	Activation
Input	-	(3, 32, 32)	-	-	-	-
1	BatchNorm2d	(3, 32, 32)	-	-	-	-
	HebbianConv2d	(96, 28, 28)	5x5	1	0	Cosine
	Triangle	(96, 28, 28)	-	-	-	power=0.7
	MaxPool2d	(96, 14, 14)	2x2	2	0	-
2	BatchNorm2d	(96, 14, 14)	-	-	-	-
	HebbianDepthConv2d	(96, 12, 12)	3x3	1	0	Cosine
	BatchNorm2d	(96, 12, 12)	-	-	-	-
	HebbianConv2d	(384, 12, 12)	1x1	1	0	Cosine
	Triangle	(384, 12, 12)	-	-	-	power=1.4
3	BatchNorm2d	(384, 12, 12)	-	-	-	-
	HebbianDepthConv2d	(384, 10, 10)	3x3	1	0	Cosine
	BatchNorm2d	(384, 10, 10)	-	-	-	-
	HebbianConv2d	(1536, 10, 10)	1x1	1	0	Cosine
	Triangle	(1536, 10, 10)	-	-	-	power=1.0
	AvgPool2d	(1536, 5, 5)	2x2	2	0	-
Output	Linear	(10)	-	-	-	-

Table 11: Net_Depthwise_Residual CNN Architecture

Layer	Type	Output Shape	Kernel	Stride	Padding	Activation
Input	-	(3, 32, 32)	-	-	-	-
1	BatchNorm2d	(3, 32, 32)	-	-	-	-
	HebbianConv2d	(96, 32, 32)	5x5	1	2	Cosine
	Triangle	(96, 32, 32)	-	-	-	power=0.7
	MaxPool2d	(96, 16, 16)	4x4	2	1	-
2	HebbianResidualBlock	(384, 16, 16)	-	-	-	power=1.4
	MaxPool2d	(384, 8, 8)	4x4	2	1	-
3	HebbianResidualBlock	(1536, 8, 8)	-	-	-	power=1.0
	AvgPool2d	(1536, 4, 4)	2x2	2	0	-
Output	Linear	(10)	-	-	-	-
HebbianResidualBlock Internal Structure						
Main Path	BatchNorm2d	(in_ch, H, W)	-	-	-	-
	HebbianConv2d	(hidden_dim, H, W)	1x1	1	0	Cosine
	Triangle	(hidden_dim, H, W)	-	-	-	power=act
	BatchNorm2d	(hidden_dim, H, W)	-	-	-	-
	HebbianDepthConv2d	(hidden_dim, H, W)	3x3	1	1	Cosine
	Triangle	(hidden_dim, H, W)	-	-	-	power=act
	BatchNorm2d	(hidden_dim, H, W)	-	-	-	-
	HebbianConv2d	(out_ch, H, W)	1x1	1	0	Cosine
Shortcut	BatchNorm2d*	(in_ch, H, W)	-	-	-	-
	HebbianConv2d*	(out_ch, H, W)	1x1	1	0	Cosine
	Add	(out_ch, H, W)	-	-	-	-
	Triangle	(out_ch, H, W)	-	-	-	power=act

* Only if in_channels $\neq$ out_channels