

# A Bio-inspired Convolutional Layer

Code at <https://github.com/Einlar/biopytorch>

---

FRANCESCO MANZALI

University of Padova

August 22, 2021

## 1 Bio-Learning

---

- Ideas
- Algorithm
- Issues

## 2 Convolutions

---

- Properties
- Bio-Convolutions
- Algorithm
- Hyperparameters

## 3 Benchmark

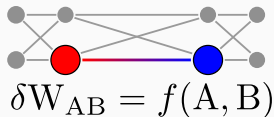
---

- Dataset
- Architecture
- Results

## 1. Local Learning Rule

The change of a weight depends only on the activations of the neurons it connects.

- Only a **forward** pass is needed for training.
- Every layer except the very last is trained in an **unsupervised** way. The last layer can use the **delta rule** for a supervised task.

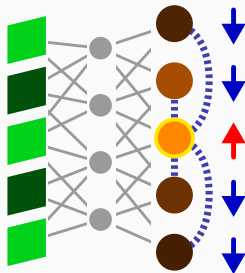


“Bio-inspired” algorithm by Krotov, Hopfield, *Unsupervised learning by competing hidden units*, 2019 [1].

## 2. Competition of Neurons

A neuron which is strongly activated for a pattern reduces the activation of all the other neurons in the same layer (**lateral inhibition**).

- Neurons “specialize” on **different** patterns.
- Only few neurons are active at a given time (**sparse** activation)

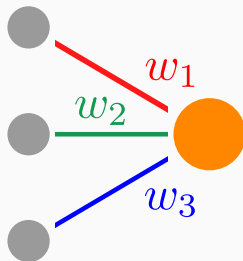


### 3. Normalization of Weights

Weights cannot grow indefinitely, and tend to the surface of a  $p$ -norm sphere.

$$|w_1|^p + |w_2|^p + |w_3|^p \rightarrow R^p$$

- A lower  $p$  forces most weights to be small/zero (**sparse** weights).



Let's inspect how the algorithm works through an **example**:

## Input Samples

Consider the following batch of 3 samples  $\{\mathbf{v}_b\}_{i=1,2,3}$ , each a vector with 4 entries.

Each sample is a **row** in the batch matrix  $V$ .

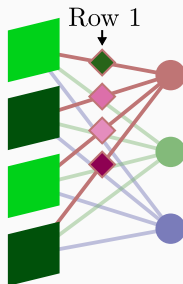
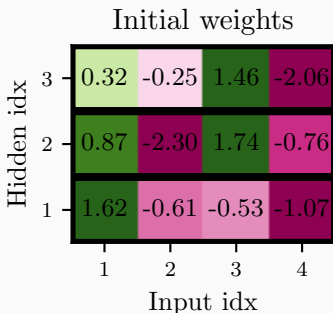
Batch	
Batch idx	
3	0.40 0.10 0.10 0.40
2	0.10 0.30 0.30 0.10
1	0.10 0.20 0.30 0.40
	1 2 3 4
	Input dim.

Consider 3 hidden neurons. Each of them has 4 weights connecting it to the 4 available inputs.

The weights of a neuron are a **row** in the weights matrix  $W$ .

Entries are normally distributed:

$$W_{\mu i} \sim \mathcal{N}(0, 1)$$



For any sample  $\mathbf{v}_b$ , compute the **current** received by the  $\mu$ -th neuron as:

$$I_{\mu b} = \langle \mathbf{W}_{\mu}, \mathbf{v}_b \rangle_p \equiv \sum_{\nu=1}^n |W_{\mu\nu}|^{p-1} \text{sgn}(W_{\mu\nu}) V_{\nu b}$$

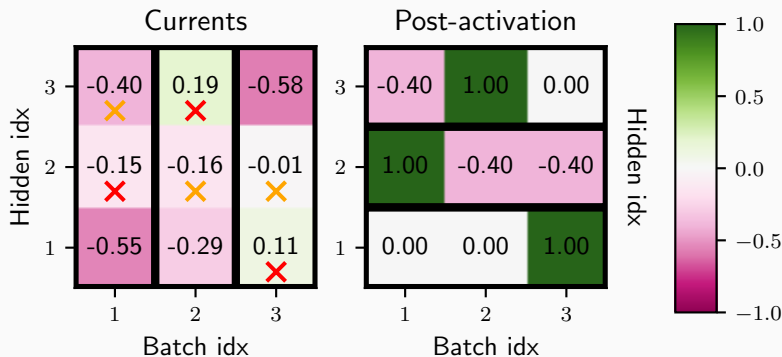
Then neurons **compete** with each other for their *post-* activations  $g(I_{\mu\nu})$ .

For each sample:

- The neuron with the **highest** current “wins”, and gets a positive post-activation, which will push it *towards* that sample.
- The neuron with the  $k$ -th highest current “loses”, and gets a negative post-activation. In this way, at most  $k - 1$  neurons can be “close” to a pattern.

$$g(I_{\mu b}) = \begin{cases} 1 & \mu = r_{1b} \\ -\Delta & \mu = r_{k\mu} \\ 0 & \text{otherwise} \end{cases} \quad r_{sb} = \begin{array}{l} \text{index of the neuron with the } s\text{-th} \\ \text{highest current for the } b\text{-th sample} \end{array}$$

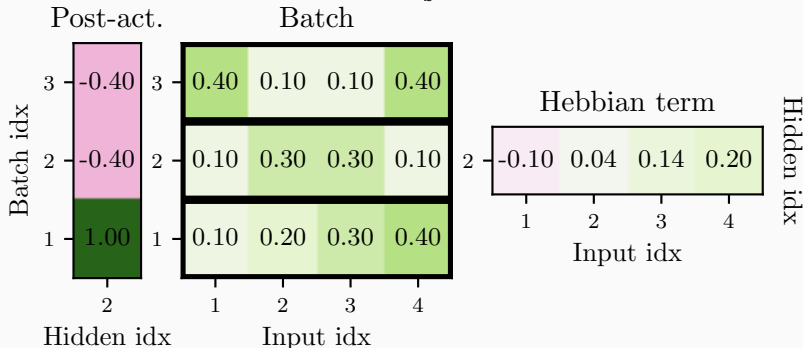




**Figure 2** – Currents  $I_{\mu b}$  and post-activations  $g(I_{\mu b})$  for  $p = 2, k = 2, \Delta = 0.4$ . For each sample in the batch (column in the left figure), the highest current is marked with a red  $\times$ , and the  $k$ -highest with an orange  $\times$ .

The **hebbian** term for the change of weights is given by:

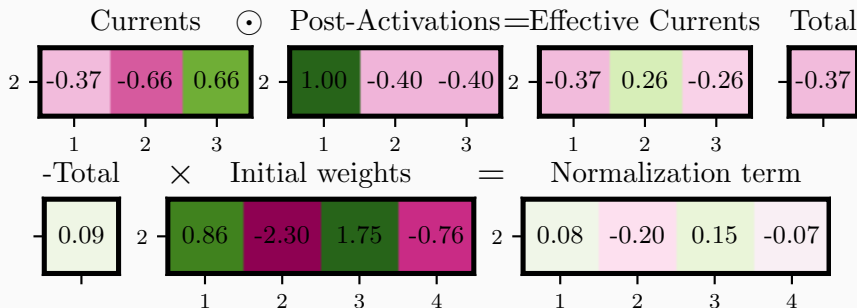
$$\Delta W_{\mu i, \text{hebb.}} = \sum_b g(I_{\mu b}) V_{ib}$$



**Figure 3** – To get the hebbian term, multiply every activation (left) by the corresponding sample (rows of center matrix). Then sum the rows. The figure shows the computation for  $\mu = 2$ .

The normalization term is given by:

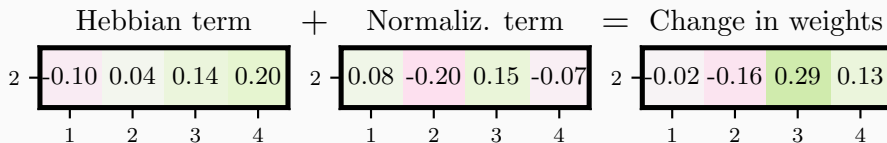
$$\Delta W_{\mu i, \text{norm.}} = - \left( \sum_b g(I_{\mu b}) I_{\mu b} \right) W_{\mu i}$$



**Figure 4** – Multiply the currents by the post-activations to get the effective currents, and sum the results to get the total. This is then used to multiply the initial weights to get the normalization term (for  $\mu = 2$  here).

The change in weights  $\Delta W_{\mu i}$  consists of two terms: **hebbian** and **normalization**:

$$\Delta W_{\mu i} = \Delta W_{\mu i, \text{hebb.}} + \Delta W_{\mu i, \text{norm.}}$$



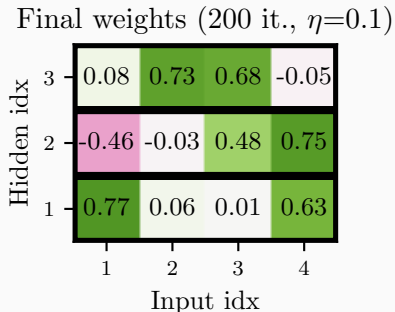
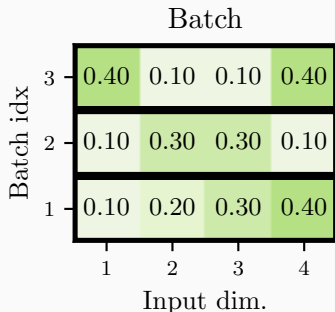
**Figure 5** – Proposed change of weights for the second hidden unit ( $\mu = 2$ ) in the above example.

In practice, weights are updated iteratively, choosing a different minibatch at each iteration:

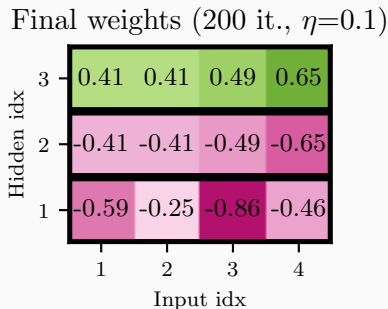
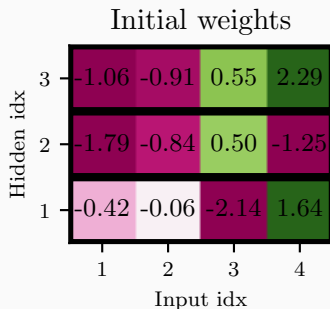
$$W_{\mu i}^{(t+1)} = W_{\mu i}^{(t)} + \eta_t \Delta W_{\mu i}^{(t)} \quad W_{\mu i}^{(0)} \sim \mathcal{N}(0, 1)$$

where  $\eta_t$  is the (time-dependent) learning rate.

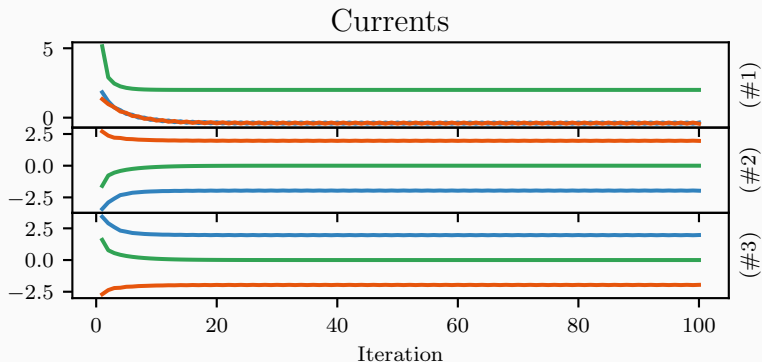
After 200 iterations with  $\eta = 0.1$ , the final weights converge:



However, a different weight initialization does **not** converge:



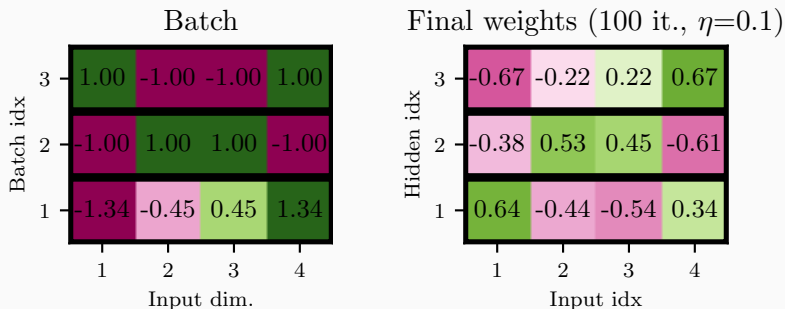
**Note:** the weights of the third neuron ( $\mu = 3$ ) are closer to the batch mean (positive), and so this neuron “dominates” over all the others, preventing them from adjusting with Hebbian learning.



**Figure 8** – Evolution of the currents during training. Each subplot refers to a different input sample, with the **blue**, **orange** and **green** lines referring respectively to the **first**, **second** and **third** neuron. Note how the current for  $\mu = 3$  (the green one) is always the highest.

To solve this issue, we would like the columns of  $I_{\mu b}$  to have (approximately) independent rankings.

This is done by normalizing the rows of  $V_{bi}$ :

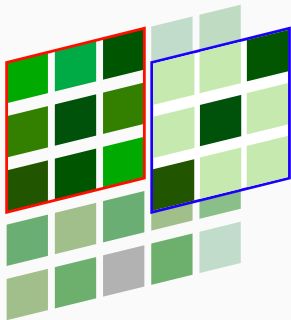


**Figure 9** – The same weight initialization from before can converge if the batch samples are properly normalized to 0 mean and 1 std.

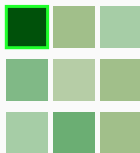


The main properties of a **convolutional layer**:

- **Local Receptive Fields**: each neuron receives inputs only from a **patch** of the image.
- **Weight sharing**: the same set of weights (**kernel**) is used for all the receptive fields.

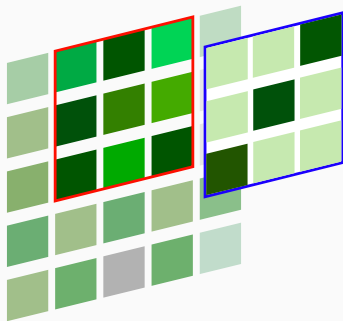


Output



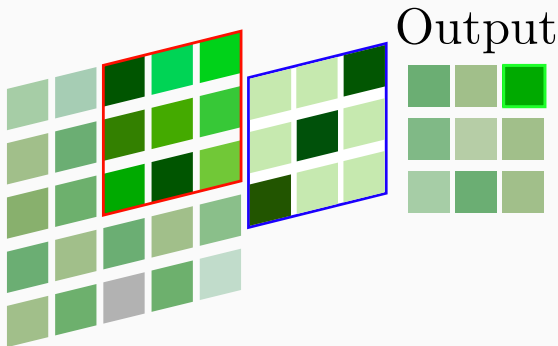
The main properties of a **convolutional layer**:

- **Local Receptive Fields**: each neuron receives inputs only from a **patch** of the image.
- **Weight sharing**: the same set of weights (**kernel**) is used for all the receptive fields.



The main properties of a **convolutional layer**:

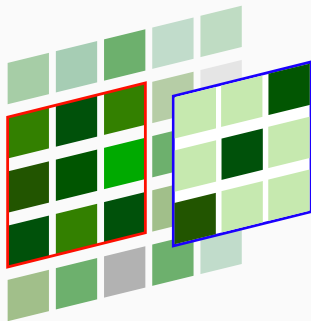
- **Local Receptive Fields**: each neuron receives inputs only from a **patch** of the image.
- **Weight sharing**: the same set of weights (**kernel**) is used for all the receptive fields.



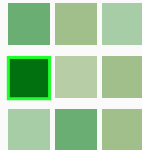
# Convolutions Layer

The main properties of a **convolutional layer**:

- **Local Receptive Fields**: each neuron receives inputs only from a **patch** of the image.
- **Weight sharing**: the same set of weights (**kernel**) is used for all the receptive fields.



Output



## Convolutional Layers > Fully-Connected Layers

(for image classification)

### Local Receptive Fields

- Patterns in images are mostly **local**.
- Different **scales** can be captured by tweaking the kernel size, or by rescaling the inputs (e.g. through maxpooling).
- Inspired by retinal ganglion cells and V1 neurons in the primary visual cortex.

### Weight sharing

- Ability to detect the **same** pattern in **different** position in the image.
- Images can be described by few “universal” patterns that can appear in any position.
- **Fewer weights** to learn improve performance and help against overfitting.

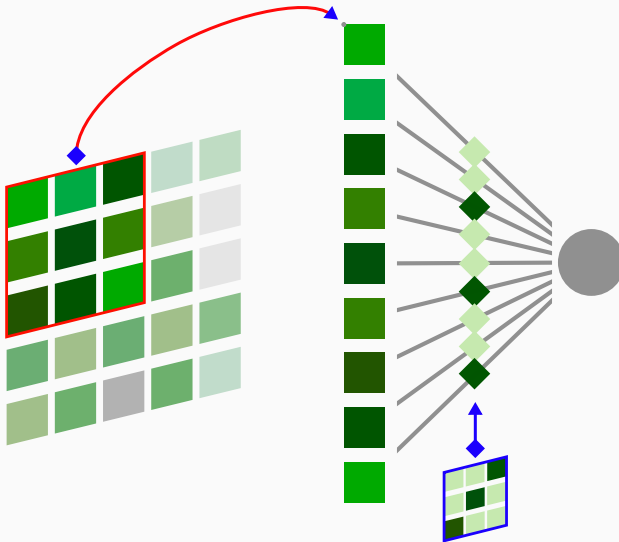
How to *learn* Convolutional Layers with  
the bio-inspired rule?

# How to *learn* Convolutional Layers with the bio-inspired rule?

We just need the right perspective!

# Convolutions Equivalence

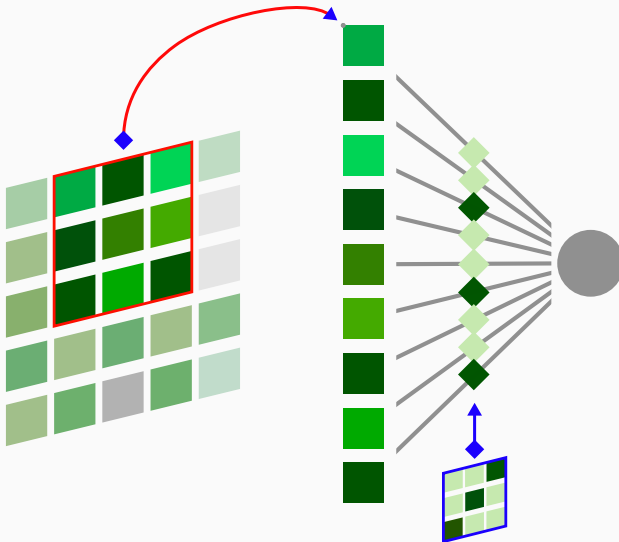
A **convolutional kernel** is just a **perceptron** working on patches:





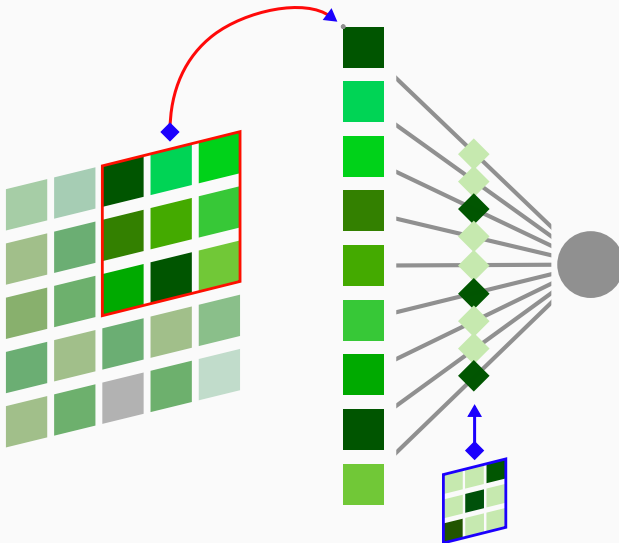
# Convolutions Equivalence

A **convolutional kernel** is just a **perceptron** working on patches:



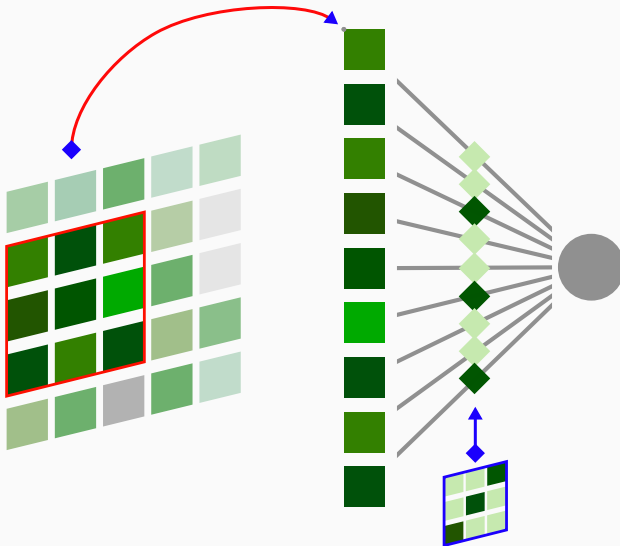
# Convolutions Equivalence

A **convolutional kernel** is just a **perceptron** working on patches:

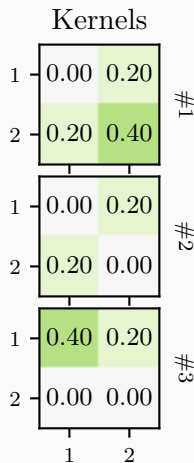
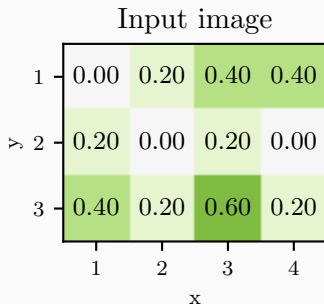


# Convolutions Equivalence

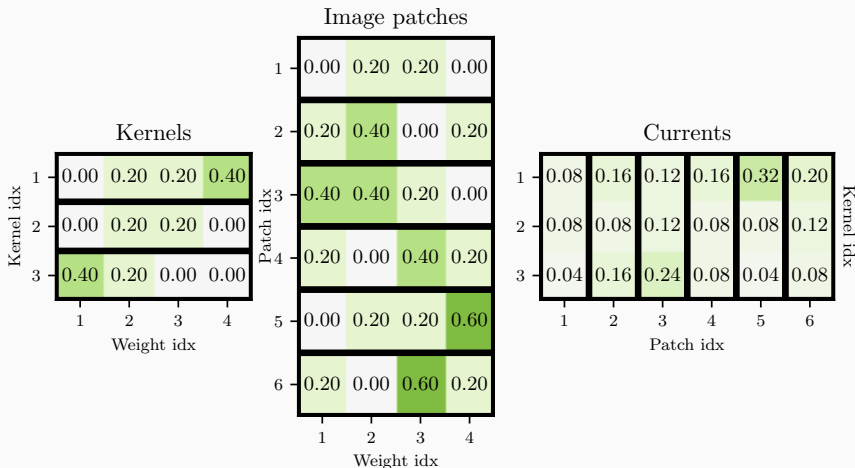
A **convolutional kernel** is just a **perceptron** working on patches:



Consider more **kernels**, acting on a one-channel image:

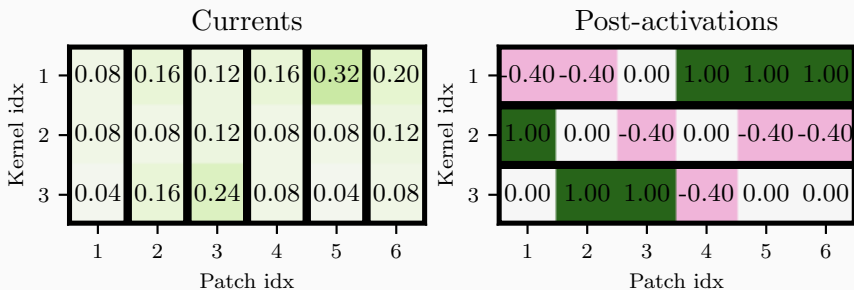


By “**unfolding**” patches and kernels, the setup is the same as before:



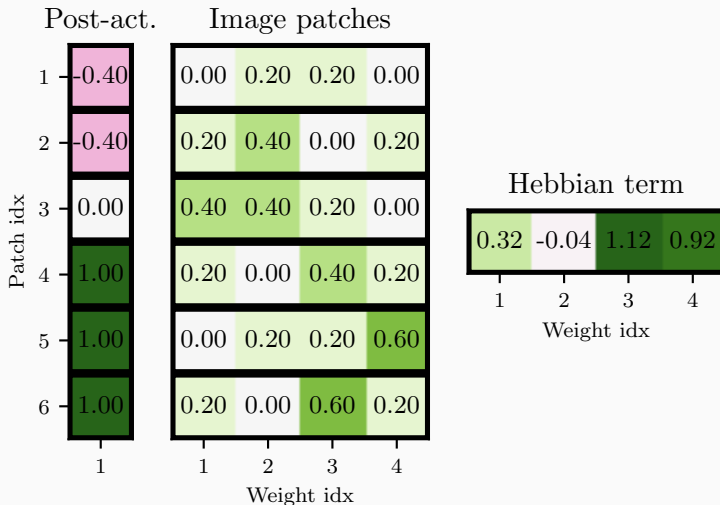
**Figure 11** – Currents are the scalar product of each row of the “unfolded” kernels and each patch. This is the (reshaped) output of a 2d convolution.

Kernels compete as **neurons**:



**Figure 12** – The highest value in each column receives a (post)activation of 1. The  $k$ -th highest gets instead a  $-\Delta$ , while all the others are set to 0. Here  $\Delta = 0.4$  and  $k = 2$ .

The **Hebbian term** is just the sum of the patches, weighted by the (post)activations:



### ! **Unfolding** patches is highly inefficient!

- **Many patches:** the number of patches scales as the square of the image size. Memory is quickly filled with a lot of **redundant** information.

*For a  $32 \times 32$  image (CIFAR-10), with a  $5 \times 5$  kernel, there are  $28^2 = 784$  patches!*

- **Not optimized:** modern CUDA kernels use many optimizations for 2d convolutions (e.g. Fourier transforms).

As a result, the above algorithm is *slow* and usually *crashes* due to the limited memory of a GPU.



### ! Unfolding patches is highly inefficient!

- **Many patches:** the number of patches scales as the square of the image size. Memory is quickly filled with a lot of **redundant** information.

*For a  $32 \times 32$  image (CIFAR-10), with a  $5 \times 5$  kernel, there are  $28^2 = 784$  patches!*

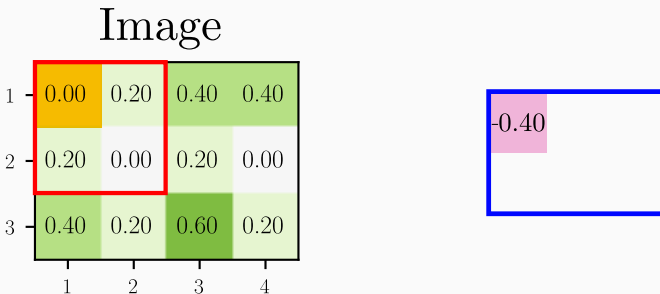
- **Not optimized:** modern CUDA kernels use many optimizations for 2d convolutions (e.g. Fourier transforms).

As a result, the above algorithm is *slow* and usually *crashes* due to the limited memory of a GPU.

But it can be rewritten using a 2d convolution!

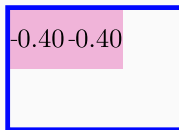
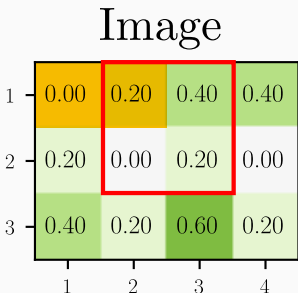
Consider the first entry of the Hebbian term.

This is the scalar product between the **post-activations** vector and the **first column** of the image **patches**, which contains the **first** value of each receptive field:



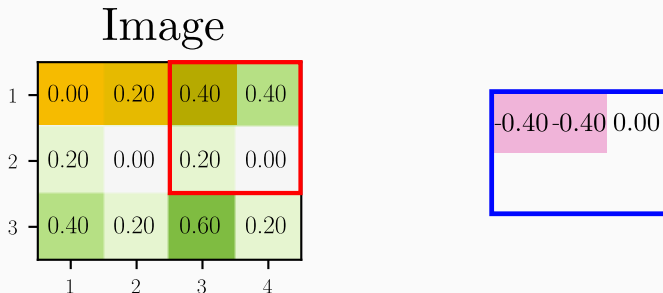
Consider the first entry of the Hebbian term.

This is the scalar product between the **post-activations** vector and the **first column** of the image **patches**, which contains the **first** value of each receptive field:



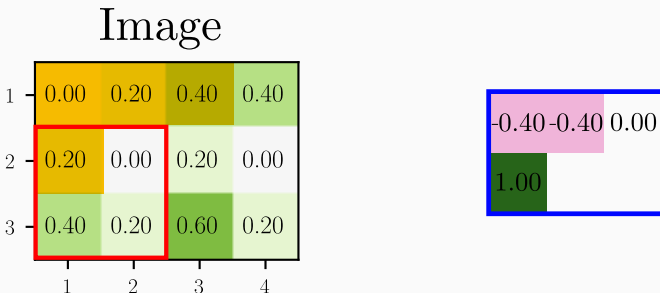
Consider the first entry of the Hebbian term.

This is the scalar product between the **post-activations** vector and the **first column** of the image **patches**, which contains the **first** value of each receptive field:



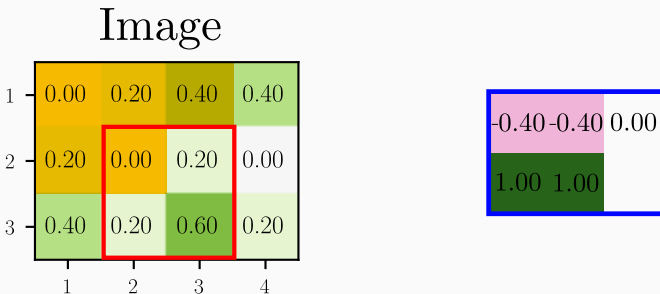
Consider the first entry of the Hebbian term.

This is the scalar product between the **post-activations** vector and the **first column** of the image **patches**, which contains the **first** value of each receptive field:



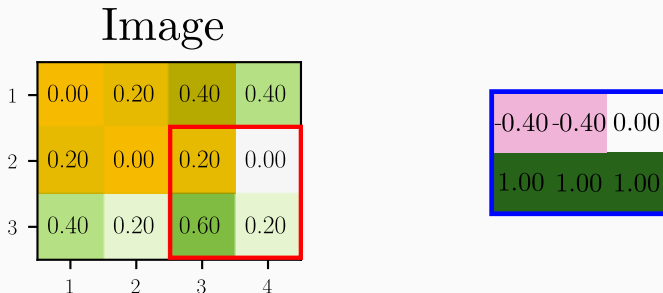
Consider the first entry of the Hebbian term.

This is the scalar product between the **post-activations** vector and the **first column** of the image **patches**, which contains the **first** value of each receptive field:



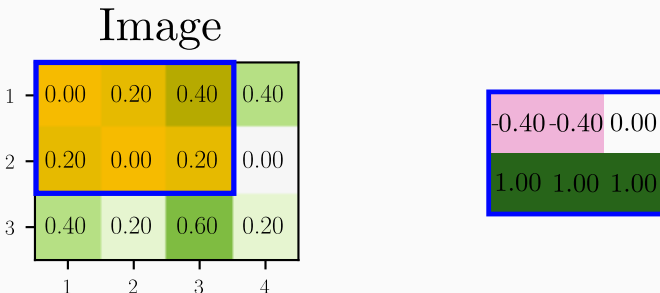
Consider the first entry of the Hebbian term.

This is the scalar product between the **post-activations** vector and the **first column** of the image **patches**, which contains the **first** value of each receptive field:



Consider the first entry of the Hebbian term.

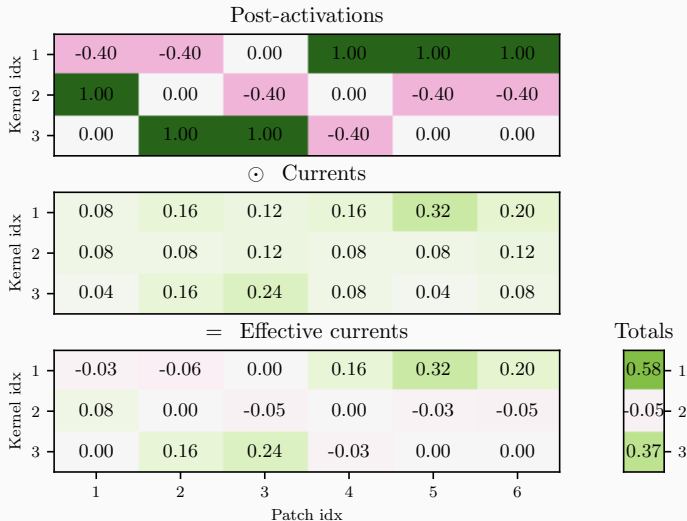
This is the scalar product between the **post-activations** vector and the **first column** of the image **patches**, which contains the **first** value of each receptive field:

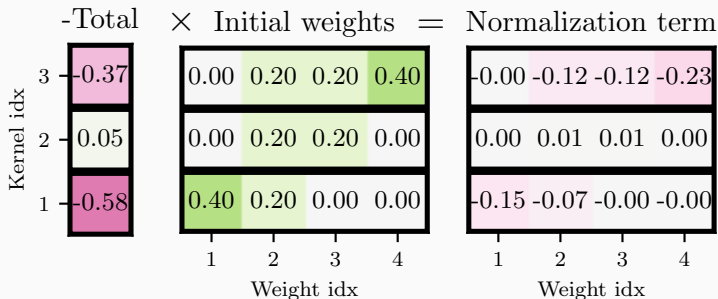


This is a 2d convolution with a  $2 \times 3$  kernel, obtained by *reshaping* the post-activations!



For the **normalization term**, the procedure is the same as before:





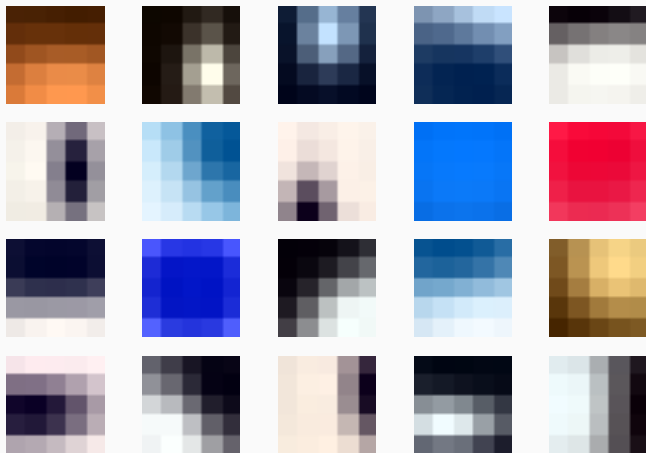
This is just a multiplication (with broadcasting), so it does not need any optimization.

### What about the **implementation**?

- Implemented with PyTorch 1.9 [2].
- Supports training on GPU, and all the parameters of a PyTorch `nn.Conv2d` (stride, dilation, groups...).

### What about **performance**?

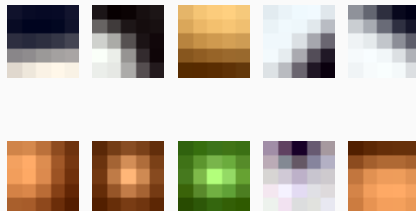
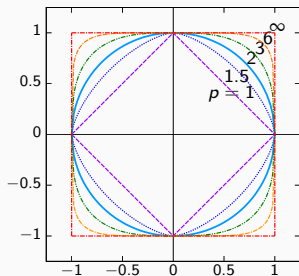
- Each training pass requires **two** 2d-convolutions on the input tensor, with the first one being the forward pass (plus some faster operations).
  - $2 - 3\times$  slower than a `Conv2d` forward.
  - Up to  $5\times$  faster than a “unfold” implementation (e.g. `dcasbol/biolearn_torch`), with significantly less memory required!
- Few epochs ( $< 5$ ) are needed to converge if the hyperparameters ( $p, k, \Delta, \eta$ ) are chosen well.



**Figure 20** – First 20  $5 \times 5$  filters (out of 96) after training for 2 epochs on 40k RGB samples from CIFAR-10, with  $k = 2$ ,  $p = 5$  and  $\Delta = 0.2$ . Learning rate starts at .007 and decays exponentially to  $10^{-4}$  with  $\lambda = .8$ .

## 1. Lebesgue norm $p$

- Lower  $p$  means **sparser** weights (fewer weights in a kernel can be significantly  $\neq 0$ ).
- Low  $p$  leads generally to *more localized* filters (e.g. edges/corners). High  $p$  leads to *diffuse* filters (e.g. on-center, gradients).
- **Convergence** ( $p$ -norm  $\rightarrow 1$ ) is best for *middle* values:  $p = 5, 6$ .



(a) Filters with  $p = 3$  (**top**) and  $p = 10$  (**bottom**), with  $k = 2$  and  $\Delta = .2$ .

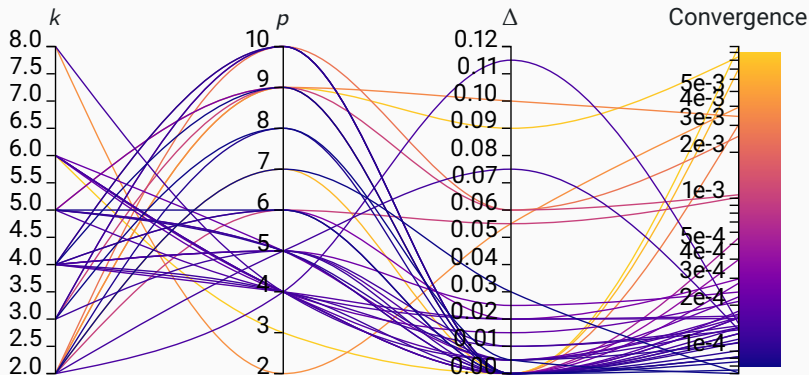
## 2. Ranking param $k$

- The neurons receiving the  $k$ -th highest currents are *pushed away* from those patterns.
- Thus, the number of neurons *encoding* a similar pattern is  $\leq k - 1$ .
- Lower  $k$  leads to sparser activations (i.e. fewer kernels are active for a given sample).
- Convergence ( $p$ -norm  $\rightarrow 1$ ) is best for  $k = 4, 5$ . A lower  $k = 2, 3$  works with higher  $\Delta$ .

## 3. Anti-Hebbian strength $\Delta$

- Specifies how much *losing* neurons are pushed away from a pattern.
- Higher values of  $\Delta$  can generate more *diverse* patterns, but if  $\Delta$  is too high, the training does not converge.
- Convergence ( $p$ -norm  $\rightarrow 1$ ) is very difficult for  $\Delta > .2$ , and is generally easier for  $\Delta \leq .1$ .

## BioConv2d - Hyperparameters Optimization



**Figure 22** – Convergence is measured as  $\max_{\mu} \left| \sum_i |W_{\mu i}|^p - 1 \right|$  (since  $\sum_i |W_{\mu i}|^p \rightarrow R^p = 1$ ).  
 ~ 500 trials are done on CIFAR-10 through Optuna [3], and the results are plotted with Wandb. Only “converged” runs (convergence  $\leq .01$ ) are shown.

## CIFAR-10 Dataset [4]

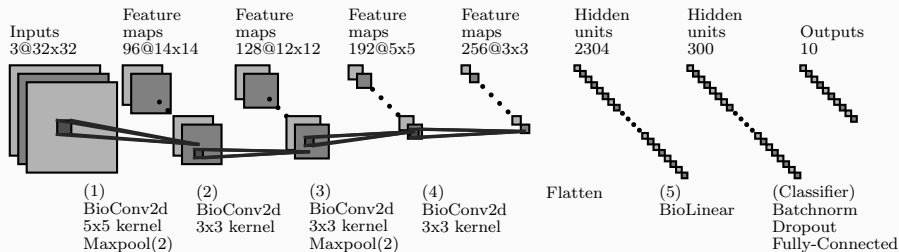
- 60k  $32 \times 32$  RGB images of 10 classes, divided in 40k images for training, 10k for validation and 10k for testing.
- Initial **normalization** to  $[0, 1]$  range, by dividing pixel values by 255.
- **Data augmentation**: “Reflect” padding of 4 + Random Crop, Random horizontal flips ( $p = .5$ ).



**Figure 23** – CIFAR-10 classes.



Common **architecture**, taken from [5]:



**Figure 24** – The first 5 layers (4 convolutional, one linear) are trained using the *bio-learning* rule, while the last layer (classifier) is trained with SGD. Each hebbian layer is preceded by a **batch normalization** (without affine transformation, so with no learnable parameters), and followed by a **ReLU** activation.

The **classifier** can be *adapted* to the output of each intermediate layer, to inspect the change in performance given by adding each layer.

The following hyperparameters are kept **fixed**:

- **Batch size**: 64.
- **Hebbian learning rate**
  - BioConv2d layers: start at .007, decay exponentially to  $10^{-4}$  with  $\lambda = 0.8$ .
  - BioLinear layer: start at .1, decay exponentially to .005 with  $\lambda = .1$ .
- Hebbian layers are trained until they reach convergence  $\leq .01$  at the end of an epoch, or if convergence starts increasing for more than 10 batches.
- Training of Hebbian layers can be either **sequential** (wait for convergence of one layer to train the next), or **all at once** (train all layers on the same batches).
- **Optimizer** (classifier): SGD with momentum = 0.9, learningrate starting at 0.05, with Cosine Annealing with Warm Restarts ( $T_0 = 10$ ,  $T_{\text{mult}} = 2$ ) [6].

## Performance and best hyperparameters

#layers	1	2	3	4	5
Accuracy (val)	66.97	65.44	63.87	58.79	45.91
Accuracy (test)	66.21	65.22	63.08	58.86	45.45
$p$	2	8	8	8	8
$k$	9	3	5	7	2
$\Delta$	.08	.34	.25	.235	.335
Dropout	.2	.25	.05	.1	.1
Params	195k	302k	387k	804k	1.475M

**Table 1** – Results of hyperparameter optimization (Optuna,  $\sim 100$ -200 trials for each case). Note how higher  $p$  and  $\Delta$  are necessary for deeper architectures, but with lower dropout. However, hebbian learning quickly loses performance when adding layers, because the learning rule has “no way to know” which features should be kept for improving classification.

## Test accuracy with different learning rules

#layers	1	2	3	4	5
Hebbian (Krotov)	<b>66.21</b>	65.22	63.08	58.86	45.45
Hebbian ([5])	63.92	63.81	58.28	52.99	41.78
Full SGD ([5])	60.71	<b>66.30</b>	<b>72.39</b>	<b>82.69</b>	<b>84.95</b>

**Table 2** – Comparison with results from [5]. Krotov's learning rule seems slightly better. Moreover, contrary to [5], there is no “teaching signal” added to neurons in the unsupervised layer to develop class-specificity. Nonetheless, a network fully trained with SGD can achieve higher performance, especially for deeper layers.

- **Convolutions** work better!
  - The network with 1 fully-connected unsupervised layer from [1] achieved only 55.26% accuracy, but with 6.166M parameters.
  - The convolutional analogue achieves 66.21% accuracy, with only 195k parameters.
- Hebbian layers requires  $< 5$  epochs to train (**very fast**).
- Performance is very good (better than SGD!) for shallow networks, but decreases for deeper ones.
  - Possible **applications**: transfer learning, fine-tuning, weight initialization.
  - Supervised “top-down” signals are necessary to solve the performance lost to depth.

- [1] Dmitry Krotov and John J. Hopfield.  
**Unsupervised learning by competing hidden units.**  
*Proceedings of the National Academy of Sciences*, 116(16):7723–7731, 2019.
- [2] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala.  
**Pytorch: An imperative style, high-performance deep learning library.**  
In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.
- [3] Takuya Akiba, Shotaro Sano, Toshihiko Yanase, Takeru Ohta, and Masanori Koyama.  
**Optuna: A Next-generation Hyperparameter Optimization Framework.**  
*arXiv:1907.10902 [cs, stat]*, July 2019.  
arXiv: 1907.10902.

- [4] Alex Krizhevsky.  
**Learning Multiple Layers of Features from Tiny Images.**  
page 60.
- [5] Giuseppe Amato, Fabio Carrara, Fabrizio Falchi, Claudio Gennaro, and Gabriele Lagani.  
**Hebbian Learning Meets Deep Convolutional Neural Networks.**  
pages 324–334. September 2019.
- [6] Ilya Loshchilov and Frank Hutter.  
**SGDR: Stochastic Gradient Descent with Warm Restarts.**  
*arXiv:1608.03983 [cs, math]*, May 2017.  
arXiv: 1608.03983.